IBM InfoSphere DataStage
Version 8 Release 5

Server Job Developer's Guide

SC18-9898-03

IBM InfoSphere DataStage
Version 8 Release 5

Server Job Developer's Guide

SC18-9898-03

Note

Before using this information and the product that it supports, read the information in “Notices and trademarks” on page 337.

Contents

Chapter 1. IBM InfoSphere DataStage Server Jobs

InfoSphere™ DataStage® jobs consist of individual stages. Each stage describes a particular database or process. For example, one stage might extract data from a data source, while another transforms it. Stages are added to a job and linked together by using the InfoSphere DataStage and QualityStage Designer.

There are two types of stage:

- **Built-in stages**. Supplied with InfoSphere DataStage and used for extracting, aggregating, transforming, or writing data.
- **Supplemental stages**. Additional stages that can be installed in InfoSphere DataStage to perform specialized tasks that the built-in stages do not support. These include stages that are supplied as part of InfoSphere DataStage packs.

The server tool palette organizes stages into the following groups:

- **Database**. These stages read or write data that is contained in a database.
- **File**. These stages read or write data that is contained in a file or set of files.
- **Processing**. These stages perform some processing on the data that is passed through them.

The following table lists the available stage types and gives a quick guide to their function:

Type	Stage	Function
Database	ODBC (see *IBM InfoSphere DataStage and QualityStage Connectivity Guide for ODBC*)	Reads data from or writes data to databases that support the industry-standard Open Database Connectivity API.
Database	Oracle 7 Load (see *IBM InfoSphere DataStage and QualityStage Connectivity Guide for Oracle Databases*)	Generates control and data files for bulk loading data into a single table in an Oracle database.
Database	Sybase BCP Load (see *IBM InfoSphere DataStage and QualityStage Connectivity Guide for Sybase Databases* and *IBM InfoSphere DataStage and QualityStage Connectivity Guide for Microsoft SQL Server and OLE DB Data*)	Uses the BCP (Bulk Copy Program) utility to bulk load data into a single table in a Microsoft SQL Server or Sybase database.
Database	UniData® (see *IBM InfoSphere DataStage and QualityStage Connectivity Guide for IBM UniVerse and UniData*)	Reads data from or writes data to a UniData database.
Database	UniData 6 (see *IBM InfoSphere DataStage and QualityStage Connectivity Guide for IBM UniVerse and UniData*)	Reads data from or writes data to a UniData 6 database.
Database	UniVerse (see *IBM InfoSphere DataStage and QualityStage Connectivity Guide for IBM UniVerse and UniData*)	Reads data from or writes data to a UniVerse database.
File	Complex Flat File	Reads data from a complex flat file data structure.
File	Folder	Reads or writes data as files in a directory located on the InfoSphere DataStage server.
File	Hashed File	Reads data from or writes data to a hashed file.
File	Sequential File	Reads data from or writes data to a sequential file.
Processing	Aggregator	Groups incoming data and computes totals and other summary functions, then passes the data to another stage in the job.

Type	Stage	Function
Processing	Command Stage	Executes external commands, programs, and jobs.
Processing	FTP Plug-in	Reads data from or writes data to remote sequential files using FTP.
Processing	InterProcess	Provides a communication channel between two InfoSphere DataStage processes running simultaneously in the same job.
Processing	Link Collector	Combines data from multiple input links into a single output link.
Processing	Link Partitioner	Partitions data from a single input link to multiple output links.
Processing	Merge	Combines two sequential files into one or more output links.
Processing	Pivot	Maps sets of columns in an input table to a single column in an output table.
Processing	Row Merger	Merges input columns into a string and writes the string to an output column.
Processing	Row Splitter	Splits data from an input string into multiple output columns.
Processing	Sort	Sorts incoming data by ascending or descending column values and passes it to another stage in the job.
Processing	Transformer	Filters and transforms incoming data, then outputs it to another stage in the job.

General information about how to construct your job and define the required metadata by using the Designer client is in the *IBM InfoSphere DataStage and QualityStage Designer Client Guide*. This manual describes the individual stage editors that you can use when developing server jobs. Some of these stages are built-in and others are supplemental.

Supplemental Stages

There are a large number of specialized supplemental stages available for InfoSphere DataStage. These can be installed when you initially install IBM® InfoSphere DataStage, or at any time after.

Connectivity stages are used to connect to specific databases. They appear in the **Database** category on the tool palette. They are described in their respective connectivity reference guides.

Other supplemental stages are active stages that appear in the **Processing** category on the tool palette. All of these are described in this guide.

IBM InfoSphere DataStage Packs

There are a number of packs available with InfoSphere DataStage that affect server jobs, each providing a set of supplemental stages and associated functionality.

- **XML Pack**. This package is supplied with InfoSphere DataStage. It provides tools that enable you to convert data between XML documents and data tables. Features and functionality are fully described in *IBM InfoSphere DataStage XML Pack Guide*.
- **Java Pack**. This package is supplied with InfoSphere DataStage. It comprises two template stages and an API which enables you to implement InfoSphere DataStage stages in Java. It is described in *IBM InfoSphere DataStage and QualityStage Java Pack Guide*.

- **Web Services Pack**. There are two versions of the web services pack, one allows you to access web services through InfoSphere DataStage jobs, the other also allows you to publish InfoSphere DataStage jobs as Web Services. Both packages are add-ons to InfoSphere DataStage. Web Services facilities are described in *IBM InfoSphere DataStage Web Services Pack Guide*.

Custom Resources

IBM InfoSphere DataStage provides a large number of built-in transforms and routines for use in Transformer stages in server jobs. These are described in Chapter 8, "Built-In Transforms and Routines," on page 291. If you have specific requirements for custom transforms and routines, InfoSphere DataStage has a powerful procedural programming language called BASIC that allows you to define your own components. Reference material for BASIC is in Chapter 7, "BASIC Programming," on page 137. After you have developed these components, they can be reused in other InfoSphere DataStage jobs.

After Development

When you have completed the development of your IBM InfoSphere DataStage server job, you will need to compile and test it before releasing it to make it available to actually run.

InfoSphere DataStage has a debugger to help you iron out any problems with any server jobs you have designed. The debugger is described in Chapter 5, "Debugging and Compiling a Job," on page 117.

When you are satisfied with the design of the job, you can validate and run it by using the InfoSphere DataStage and QualityStage Director. You can also run jobs from another program or from the command line by using the facilities provided by the InfoSphere DataStage Development Kit, which is described in InfoSphere DataStage Development Kit (Job Control Interfaces).

Chapter 2. Optimizing Performance in Server Jobs

These topics give some design techniques for getting the best possible performance from the IBM InfoSphere DataStage jobs that you design.

Many of the topics are concerned with designing a job to run on a multiprocessor system, but there are also tips for jobs running on single processor systems.

You should read these topics before you design new jobs, but you also might want to revisit old job designs based on what you read here.

The parallel processing tips are aimed at UNIX or Windows Symmetric Multi-Processor (SMP) systems with up to 64 processors. For UNIX MPP and clustered systems (and Windows or UNIX SMP systems), parallel jobs are available. For details, see *IBM InfoSphere DataStage and QualityStage Parallel Job Developer's Guide*.

IBM InfoSphere DataStage Jobs and Processes

When you design a job you see it in terms of stages and links. When it is compiled, the server engine sees it in terms of processes that are subsequently run on the server.

How does the server engine define a process? It is here that the distinction between active and passive stages becomes important. Actives stages, such as the Transformer and Aggregator, perform processing tasks, while passive stages, such as the Sequential File stage and Hashed File stage, are reading or writing data sources and provide services to the active stages. At its simplest, active stages become processes. But the situation becomes more complicated where you connect active stages together and passive stages together.

What happens when you have a job that links two passive stages together? Obviously there is some processing going on. Under the covers InfoSphere DataStage inserts a cut-down Transformer stage between the passive stages which just passes data straight from one stage to the other, and becomes a process when the job is run.

What happens where you have a job that links two or more active stages together? By default this will all be run in a single process. Passive stages mark the process boundaries, all adjacent active stages between them being run in a single process.

The following diagrams illustrate how jobs become processes.

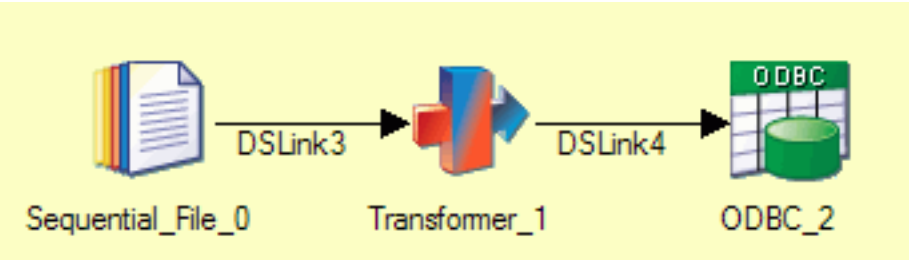


Figure 1. Single process

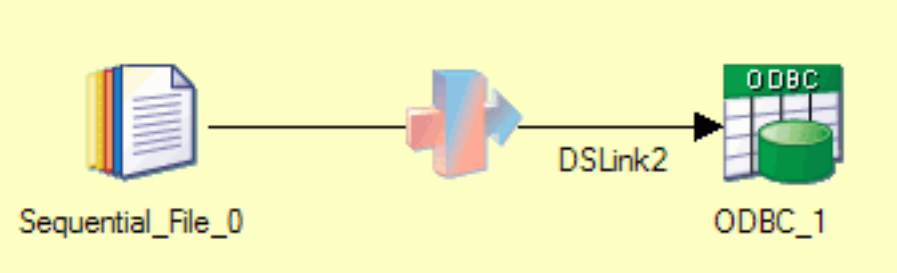


Figure 2. Single process, with a passive stage to a passive stage and an invisible Transformer stage inserted at compile time

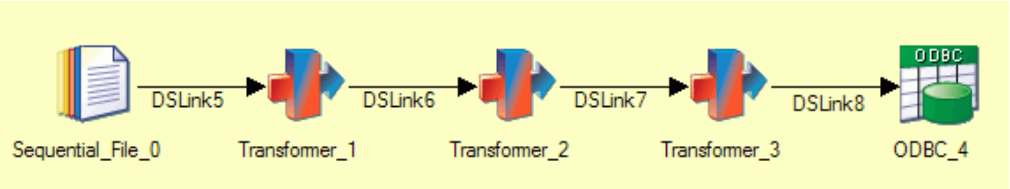


Figure 3. Single process

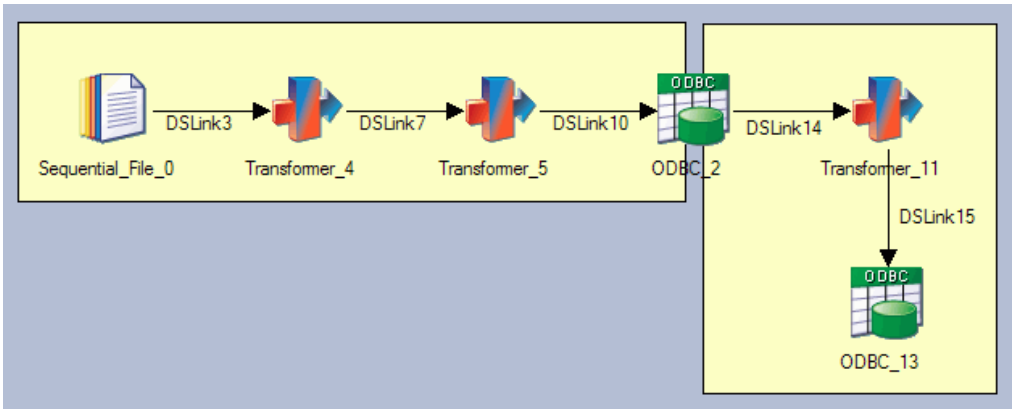


Figure 4. Two processes

Single Processor and Multi-Processor Systems

The default behavior when compiling IBM InfoSphere DataStage jobs is to run all adjacent active stages in a single process. This makes good sense when you are running the job on a single processor system. When you are running on a multiprocessor system it is better to run each active stage in a separate process so the processes can be distributed among available processors and run in parallel. The enhancements to server jobs at Release 6 of InfoSphere DataStage make it possible for you to stipulate at design time that jobs should be compiled in this way. There are two ways of doing this:

- Explicitly - by inserting InterProcess (IPC) stages between connected active stages.
- Implicitly - by turning on interprocess row buffering either project wide (using the InfoSphere DataStage and QualityStage Administrator) or for individual jobs (in the Job Properties dialog box)

The IPC facility can also be used to produce multiple processes where passive stages are directly connected. This means that an operation reading from one data source and writing to another can be divided into a reading process and a writing process able to take advantage of multiprocessor systems.

The following diagram illustrates the possible behavior for active stages:

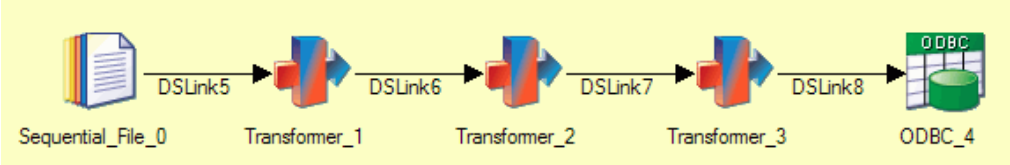


Figure 5. Default behavior

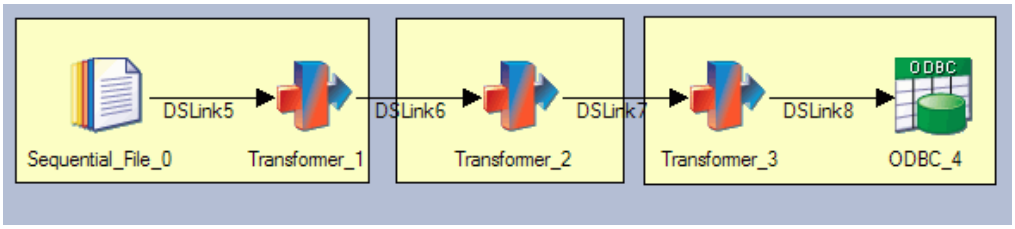


Figure 6. Implicit forcing of multiple processes via interprocess row buffering

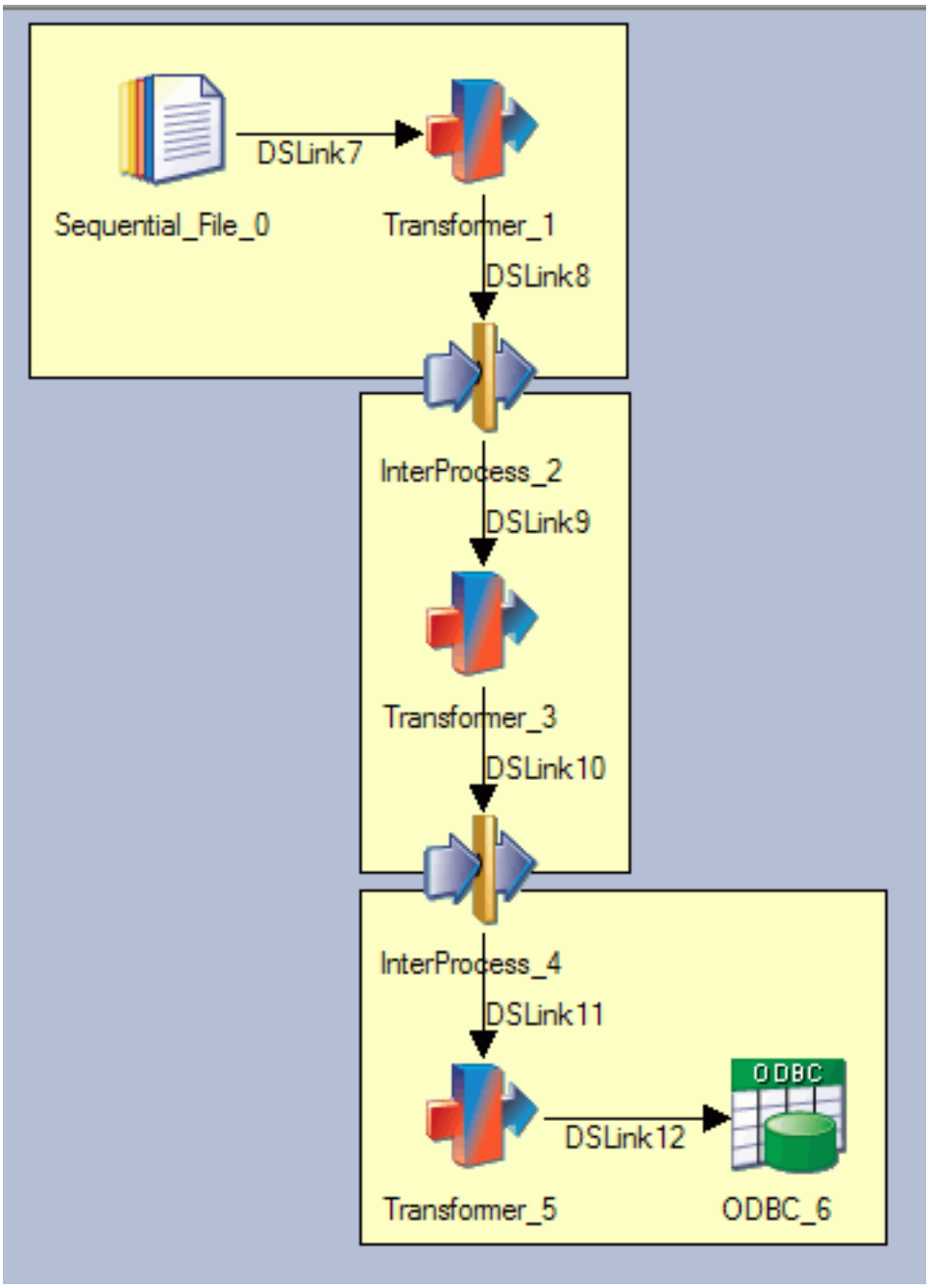


Figure 7. Using IPC stages to force multiple processes

The following diagram illustrates the possible behavior for passive stages:

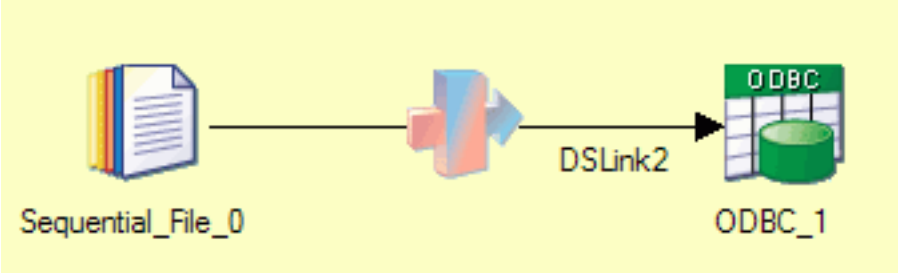


Figure 8. Default behavior, invisible Transformer stage inserted at compile time

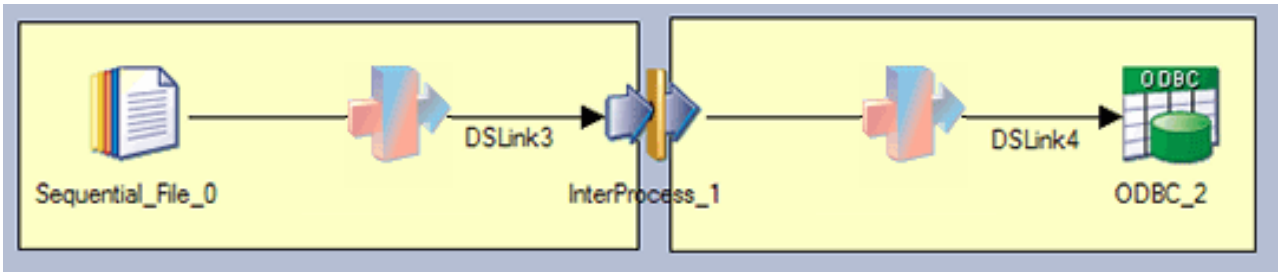


Figure 9. Using IPC stage to force multiple processes, with invisible Transformer stages inserted at compile time

Partitioning and Collecting

With the introduction of the enhanced multiprocessor support in Release 6 and later, there are opportunities to further enhance the performance of server jobs by partitioning data.

The Link Partitioner stage allows you to partition data that you are reading so it can be processed by individual processors running on multiple processors. The Link Collector stage allows you to collect partitioned data together again for writing to a single data target.

The following diagram illustrates how you might use the Link Partitioner and Link Collector stages within a job. Both stages are active, and you should turn on interprocess row buffering at the project or job level in order to implement process boundaries.

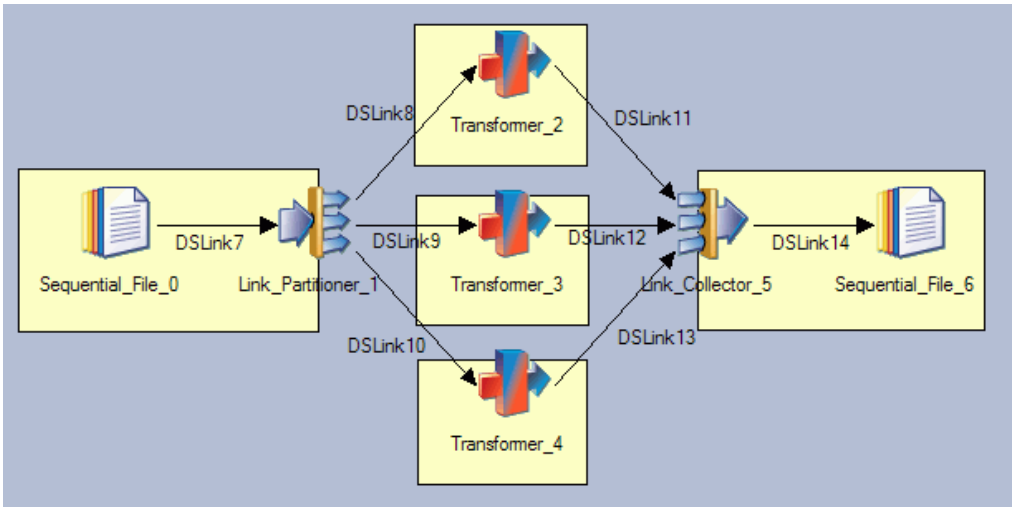


Figure 10. Using Link Partitioner and Link Collector stages

Diagnosing Job Limitations

After you design a job, you might want to run some diagnostics to see if performance can be improved.

Two factors can affect the performance of a job:

- It can be CPU limited
- It can be I/O limited

You can obtain detailed statistics about job performance to identify those parts of a job that are limiting performance and then make changes to increase performance.

The collection of performance statistics can be turned on and off for each active stage in an IBM InfoSphere DataStage job. To collect performance statistics:

1. Open the Job Run Options window:
   - In the Designer client, click the **Run** toolbar button.
   - In the Director client, select the job and click the **Run Now** toolbar button.
2. Click the **Tracing** tab.
3. Select the stages that you want to monitor in the **Stage names** list. Use shift-click to select multiple active stages.
4. Select the **Performance statistics** check box.
5. Click **Run**.

When performance tracing is turned on, a special entry is generated immediately after the stage completion message in the job log. The log entry is similar to this:

```
job.stage.DSD.StageRun Performance statistics(...)
```

To view the statistics in a tabular form, right-click the log entry and select **Detail**. You can copy the statistics in the Event Detail window and paste them into a spreadsheet to make further analysis possible.

The following diagram shows the job from which these statistics were collected. The highlighted stage is the one that has **Performance statistics** turned on:

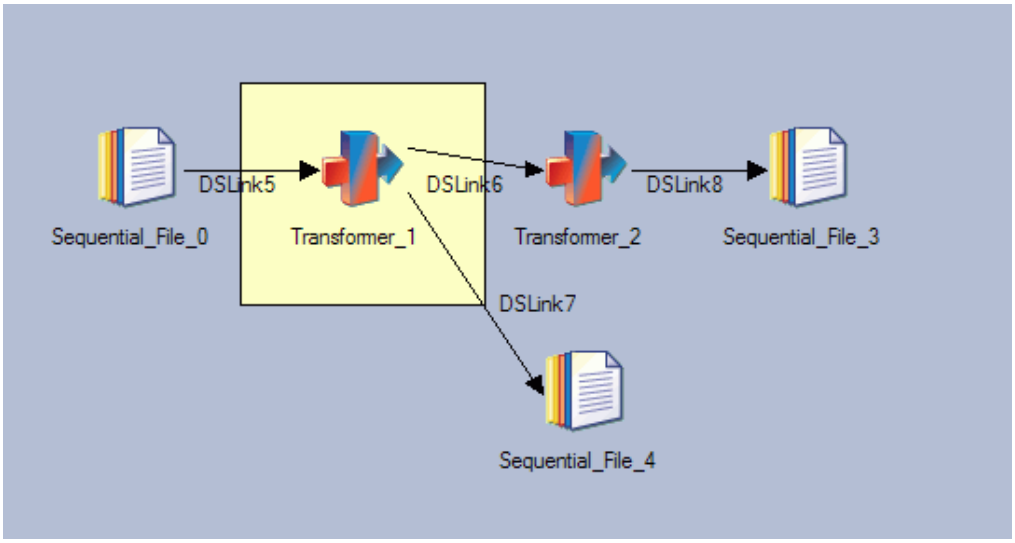


Figure 11. Sample job for performance statistics

The following table helps you interpret the statistics, which have been pasted into a spreadsheet below:

Table 1. Performance statistics

Link in job design	Action	Row in spreadsheet
DSLink5	Read Sequential File	2
DSLink6	Transformer derivation	5
	Interprocess write (Transformer stage to Transformer stage)	3
DSLink8	Write Sequential File	3
DSLink7	Transformer derivation	6
	Write Sequential File	7

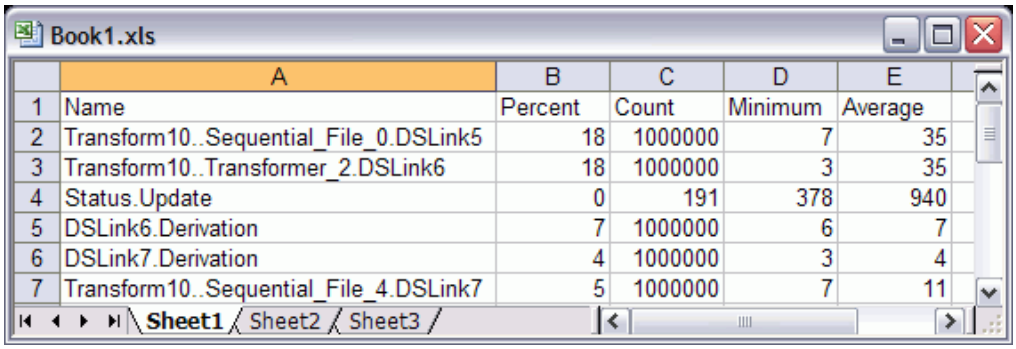


Figure 12. Performance statistics spreadsheet

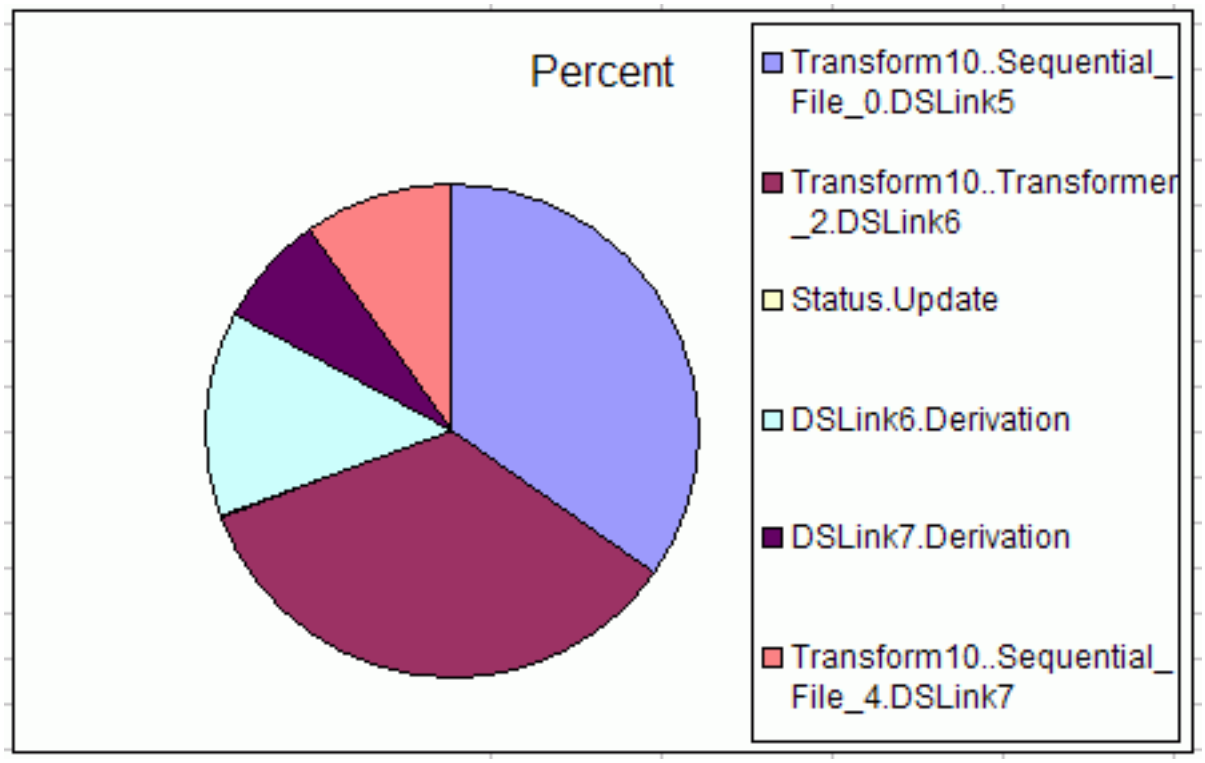


Figure 13. Graph from spreadsheet

The Stage completion log message reports the actual CPU and elapsed time used by the stage, while the Monitor view on a completed stage shows the average percentage of CPU used by that stage.

Interpreting Performance Statistics

The performance statistics relate to the per-row processing cycle of an active stage, and of each of its input and output links. The information shown is:

- **Percent**. The percentage of overall execution time that this part of the process used.
- **Count**. The number of times this part of the process was executed.
- **Minimum**. The minimum elapsed time in microseconds that this part of the process took for any of the rows processed.
- **Average**. The average elapsed time in microseconds that this part of the process took for the rows processed.

You need to take care interpreting these figures. For example, when in-process active stage to active stage links are used, the percent column will not add up to 100%. Also be aware that, in these circumstances, if you collect statistics for the first active stage, the entire cost of the downstream active stage is included in the active-to-active link (as shown in the example diagram). This distortion remains even where you are running the active stages in different processes (by having interprocess row buffering enabled) unless you are actually running on a multiprocessor system.

If the **Minimum** figure and **Average** figure are very close, this suggests that the process is CPU limited. Otherwise poorly performing jobs might be I/O limited.

If the Job monitor window shows that one active stage is using nearly 100% of CPU time, this also indicates that the job is CPU limited.

Improving Performance

The following sections give some tips on improving performance in your job designs.

CPU Limited Jobs - Single Processor Systems

You can improve the performance of most IBM InfoSphere DataStage jobs by turning in-process row buffering on and recompiling the job. This allows connected active stages to pass data via buffers rather than row by row.

You can turn in-process row buffering on for the whole project by using the Administrator client. Alternatively, you can turn it on for individual jobs by using the **Performance** tab in the Job Properties dialog box.

Note: You cannot use in-process row-buffering if your job uses COMMON blocks in transform functions to pass data between stages. It is advisable to redesign your job to use row buffering rather than COMMON blocks.

CPU Limited Jobs - Multiprocessor Systems

You can improve the performance of most IBM InfoSphere DataStage jobs on multiprocessor systems by turning on interprocess row buffering and recompiling the job. This enables the job to run using a separate process for each active stage; these will run simultaneously on separate processors.

You can turn interprocess row buffering on for the whole project by using the Administrator client. Alternatively, you can turn it on for individual jobs by using the **Performance** tab in the Job Properties dialog box.

Note: You cannot use interprocess row-buffering if your job uses COMMON blocks in transform functions to pass data between stages. It is advisable to redesign your job to use row buffering rather than COMMON blocks.

If you have one active stage using nearly 100% of CPU, you can improve performance by running multiple parallel copies of a stage process. This is achieved by duplicating the CPU-intensive stages or stages (using a shared container is the quickest way to do this) and inserting a Link Partitioner and Link Collector stage before and after the duplicated stages. The following screen shots show an example of how you might do this.

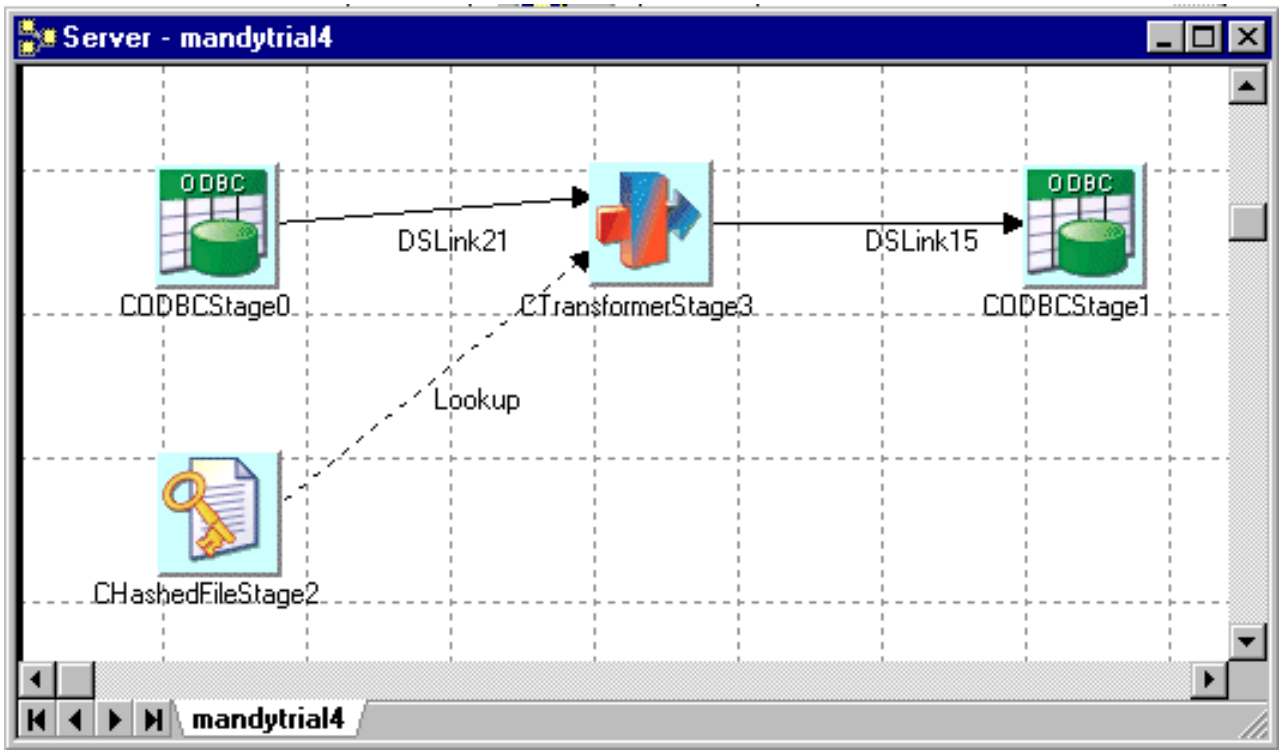


Figure 14. Example job

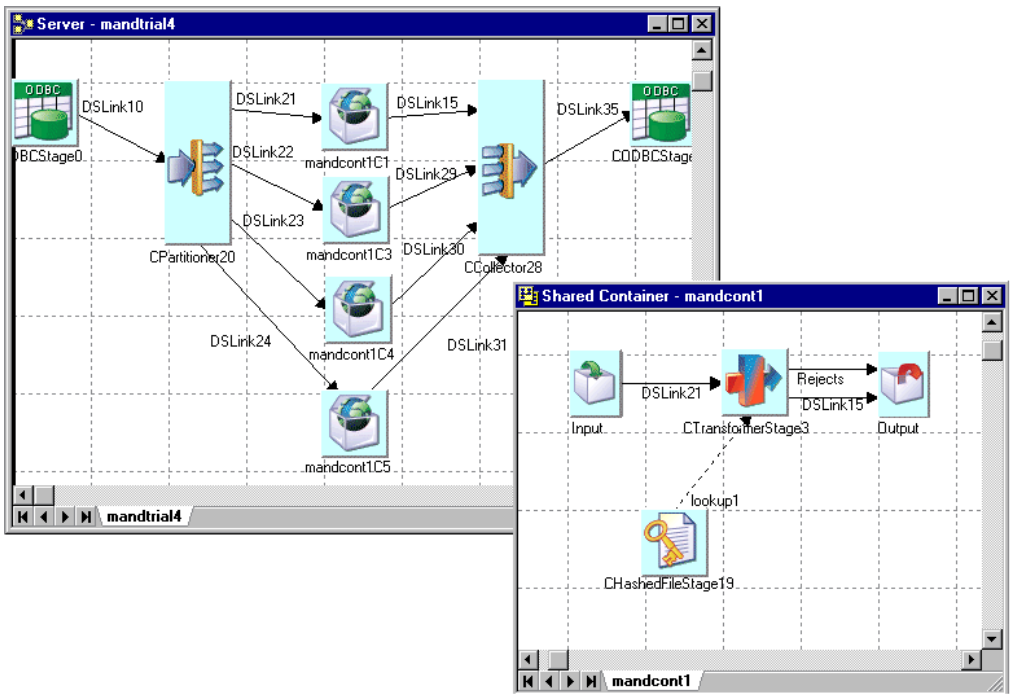


Figure 15. Example job

I/O Limited Jobs

Although it can be more difficult to diagnose I/O limited jobs and improve them, there are certain basic steps you can take:

- If you split processes in your job design by writing data to a Sequential File and then reading it back again, you can use an InterProcess (IPC) stage in place of the Sequential File stage. This will split the process and reduce I/O and elapsed time, as the reading process can start reading data as soon as it is available rather than waiting for the writing process to finish.
- If an intermediate Sequential File stage is being used to land a file so that it can be fed to an external tool, for example a bulk loader, or an external sort, it might be possible to invoke the tool as a filter command in the Sequential File stage and pass the data directly to the tool (see "Sequential File Stages" on page 43).
- If you are processing a large data set, you can use the Link Partitioner stage to split it into multiple parts without landing intermediate fields.

If a job still appears to be I/O limited after taking one or more of the above steps, you can use the performance statistics to determine which individual stages are I/O limited:

1. Run the job with a substantial data set and with performance tracing enabled for each of the active stages.
2. Analyze the results and compare them for each stage. In particular, look for active stages that use less CPU than others.

After you have identified the stage, the actions you take might depend on the types of passive stage involved in the process. Poorly designed hashed files can have particular performance implications (for help with hashed file design, see "Hash File Design" on page 15). For all stage types you might consider:

- redistributing files across disk drives
- changing memory or disk hardware

- reconfiguring databases
- reconfiguring operating system

Hash File Design

Poorly designed hashed files can be a cause of disappointing performance. Hashed files are commonly used to provided a reference table based on a single key. Performing lookups can be fast on a well-designed file, but slow on a poorly designed one. Another use is to host slowly-growing dimension tables in a star-schema warehouse design. Again, a well-designed file will make extracting data from dimension files much faster.

Basic Hash File Operation

If you are familiar with the principles of hashed files you can skip this section.

Hash files work by spreading data over a number of groups within a file. These speeds up data access as you can go to a specific group before sequentially searching through the data it contains for a particular data row. The number of groups you have, the size the groups, and the algorithm used to work out distribution is decided by the nature of the data you are storing in the file.

The rows of data are hashed (that is, allocated to groups) on a key field. The hashing algorithm efficiently and repeatably converts a string to a number in the range 1 to *n*, where *n* is the file modulus. This gives the group where the row will be written. The key field can be of any type; for example it could contain a name, a serial number, a date, and so on The type of data in the key determines the best hashing algorithm to use when writing data; this algorithm is also used to locate the data when reading it back. The aim is to use an algorithm that spreads data evenly over the file.

Another aim is to spread the data as evenly as possible over a number of groups. It is particularly important as far as performance goes not to overpopulate groups so that they have to extend into overflow groups, as this makes accessing the data inefficient. It is important to consider the size of your records (rows) when designing the file, as you want them to fit evenly into groups and not overflow.

There is a trade-off between size of group and number of groups. For most operations a good design has many groups each of small size (for example, four records per group). The sequential search for the required data row is then never that long. There might be circumstances, however, where a design would be better served by a smaller number of large groups.

IBM InfoSphere DataStage Hash Files

There are two basic types of hash file that you might use in these circumstances: static (hash) and dynamic.

- **Static Files**. These are the most performant if well designed. If poorly designed, however, they are likely to offer the worst performance. Static files allow you to decide the way in which the file is hashed. You specify:
  - **Hashing algorithm**. The way data rows are allocated to different groups depending on the value of their key field or fields.
  - **Modulus**. The number of groups the file has.
  - **Separation**. The size of the group as the number of 512-byte blocks.

    Generally speaking, you should use a static file if you have good knowledge of the size and shape of the data you will be storing in the hashed file. You can restructure a static hashed file between job runs if you want to tune it. Do this using the RESIZE command, which can be issued using the Command feature of the Administrator client. The command for resizing a static file is:

    ```
    RESIZE filename [type] [modulus] [separation]
    ```

    Where:

    filename is the name of the file you are resizing

    type specifies the hashing algorithm to use (see “Hash File Design”)

modulus specifies the number of groups in the range 1 through 8,388,608.

separation specifies the size of the groups in 512 byte blocks and is in the range 1 through 8,388,608.

- **Dynamic Files**. These are hash files which change dynamically as data is written to them over time. This might sound ideal, but if you leave a dynamic file to grow organically it will need to perform several group split operations as data is written to it, which can be very time consuming and can impair performance where you have a fast growing file. Dynamic files do not perform as well as a well-designed static file, but do perform better than a badly designed one. When creating a dynamic file you can specify the following information (although all of these have default values):
  - **Minimum modulus**. The minimum number of groups the file has. The default is 1.
  - **Group size**. The group can be specified as 1 (2048 bytes) or 2 (4096 bytes). The default is 1.
  - **Split load**. This specifies how much (as a percentage) a file can be loaded before it is split. The file load is calculated as follows:

    ```
    File Load = ((total data bytes) / (total file bytes)) * 100
    ```

    The split load defaults to 80.
  - **Merge load**. This specifies how small (as a percentage) a file load can be before the file is split. File load is calculated as for Split load. The default is 50.
  - **Large record**. Specifies the number of bytes a record (row) can contain. A large record is always placed in an overflow group.
  - **Hash algorithm**. Choose between GENERAL for most key field types and SEQ.NUM for keys that are a sequential number series.
  - **Record size**. Optionally use this to specify an average record size in bytes. This can then be used to calculate group size and large record size.

    You can manually resize a dynamic file using the RESIZE command issued using the Command feature of the Administrator client. The command for resizing a dynamic file is:

    ```
    RESIZE filename [parameter [value]]
    ```

    where:

    filename is the name of the file you are resizing.

    Parameter is one of the following and corresponds to the arguments described above for creating a dynamic file:

    ```
    GENERAL | SEQ.NUM
    MINIMUM.MODULUS  n
    SPLIT.LOAD  n
    MERGE.LOAD  n
    LARGE.RECORD  n
    RECORD.SIZE  n
    ```

By default InfoSphere DataStage will create you a dynamic file with the default settings described above. You can, however, use the **Create File** options on the Hashed File stage Inputs page to specify the type of file and its settings.

This offers a choice of several types of hash (static) files, and a dynamic file type. The different types of static files reflect the different hashing algorithms they use. Choose a type according to the type of your key, as shown below:

Type	Suitable for keys that are formed like this:
2	Numeric - significant in last 8 chars
3	Mostly numeric with delimiters significant in last 8 chars
4	Alphabetic significant in last 5 chars
5	Any ASCII significant in last 4 chars
6	Numeric significant in first 8 chars

7 Mostly numeric with delimiters significant in first 8 chars

8 Alphabetic significant in first 5 chars

9 Any ASCII significant in first 4 chars

10 Numeric significant in last 20 chars

11 Mostly numeric with delimiters significant in last 20 chars

12 Alphabetic significant in last 16 chars

13 Any ASCII significant in last 16 chars

14 Numeric whole key is significant

15 Mostly numeric with delimiters whole key is significant

16 Alphabetic whole key is significant

17 Any ASCII whole key is significant

18 Any chars whole key is significant

Operational Enhancements

There are various steps you can take within your job design to speed up operations that read and write hash files.

- **Pre-loading**. You can speed up read operations of reference links by pre-loading a hash file into memory. Specify this on the Hashed File stage Outputs page.
- **Write Caching**. You can specify a cache for write operations such that data is written there and then flushed to disk. This ensures that hashed files are written to disk in group order, rather than the order in which individual rows are written (which would by its nature necessitate time-consuming random disk accesses). If server caching is enabled, you can specify the type of write caching when you create a hash file, the file then always uses the specified type of write cache. Otherwise you can turn write caching on at the stage level via the Outputs page of the Hashed File stage.
- **Pre-allocating**. If you are using dynamic files you can speed up loading the file by doing some rough calculations and specifying the minimum modulus accordingly. This greatly enhances operation by cutting down or eliminating split operations. You can calculate the minimum modulus as follows:

  ```
  minimum modulus = estimated data size/(group size * 2048)
  ```

  When you have calculated your minimum modulus, you can create a file specifying it (using the **Create File** feature of the **Hashed File** Stage dialog box - see “Defining Hashed File Input Data” on page 40) or resize an existing file specifying it (using the RESIZE command described in “IBM InfoSphere DataStage Hash Files” on page 15).
- **Calculating static file modulus**. You can calculate the modulus required for a static file using a similar method as described above for calculating a pre-allocation modulus for dynamic files:

  ```
  modulus = estimated data size/(separation * 512)
  ```

  When you have calculated your modulus you can create a file specifying it (using the **Create File** feature of the **Hashed File Stage** dialog box - see “Defining Hashed File Input Data” on page 40) or resize an existing file specifying it (using the RESIZE command described in“IBM InfoSphere DataStage Hash Files” on page 15).

Chapter 3. Server Jobs and NLS

These topics give details about NLS in IBM InfoSphere DataStage server jobs. It covers:

- Maps and locales available in server jobs
- Loading maps and loading locales
- Considerations about character data in server jobs
- How to use maps and locales in server jobs
- Creating new maps for server jobs
- Creating new locales for server jobs

How NLS Mode Works

NLS mode works by using two types of *character set*:

- The NLS *internal character set*
- *External character sets* that cover the world's different languages

In NLS mode, InfoSphere DataStage maps between the two character sets when it's needed.

The mechanism for handling NLS differs for parallel and server jobs. They each use a different internal character set, so each uses a different set of maps for converting data. Note that it is certain types of string (that is, character) data that needs mapping, purely numeric data types never require it.

Parallel and server jobs also use different locales.

Internal Character Sets

The internal character set can represent at least 64,000 characters. Each character in the internal character set has a unique *code point*. This is a number that is by convention represented in hexadecimal format. You can use this number to represent the character in programs. InfoSphere DataStage easily stores many languages.

The NLS internal character sets conform to the Unicode standard. The Unicode consortium specify a number of ways to represent code points, called Unicode Transformation Formats (UTF). Server jobs use UTF-8, parallel jobs use UTF-16.

Because the two types of job use different internal character sets, a different set of maps are provided for conversion to and from each one (although equivalents to commonly used server job maps are provided for parallel jobs).

For more information about Unicode, see the Unicode Consortium's World Wide Web page at **http://www.unicode.org**.

Mapping

When you need to transform or transfer data, NLS maps the data to or from the external character set you want to use. NLS includes map tables for many of the character sets used in the world (see the list in *IBM InfoSphere DataStage and QualityStage Globalization Guide*). You can specify mapping at different levels within InfoSphere DataStage:

- A project-wide default. In theInfoSphere DataStage and QualityStage Administrator client you specify a default map for all server jobs in a project, and a default map for all parallel jobs in a project.

- A job default. In the InfoSphere DataStage and QualityStage Designer client, you can specify a default map used by a particular job that overrides the project default.
- A stage map. Certain parallel and server stages allow you to specify that they use a particular map. This overrides both the project default and the job detail.
- A column map. Certain parallel and server stages support per-column mapping. This allows you to specify a separate map for particular data columns. This overrides the project default, job default, and stage maps.

If your files contain only ASCII 7-bit characters, they need not be mapped.

Locales

An InfoSphere DataStage NLS *locale* is a set of *national conventions*. A locale is viewed as a separate entity from a character set. You need to consider the language, character set, and conventions for data formatting that one or more groups of people use. You define the character set independently, although for national conventions to work correctly, you must also use the appropriate character sets. For example, Venezuela and Ecuador both use Spanish as their language, but have different data formatting conventions.

Locales do not respect national boundaries. One country can use several locales, for example, Canada uses two and Belgium uses three. Several countries can use one locale, for example, a multinational business could define a worldwide locale to use in all its offices. See *IBM InfoSphere DataStage and QualityStage Globalization Guide* for a list of all the locales that are supplied with InfoSphere DataStage and the territories and languages associated with them.

Server jobs allow you to choose locales separately for several different aspects of National conventions:

- The format for times and dates
- The format for displaying numbers
- How to display monetary values
- Whether a character is alphabetic, numeric, nonprinting, and so on
- The order in which characters should be sorted (collation)

You can mix locales if required, for example you could specify times and dates in one locale and monetary conventions in another.

Parallel jobs allow you to choose locales separately for:

- The order in which characters should be sorted (collation)

You can specify locales at different levels within InfoSphere DataStage:

- A project-wide default. In the Administrator client you specify default locales for all server jobs in a project, and a default locale for all parallel jobs in a project.
- A job default. In the Designer client, you can specify default locales used by a particular job that overrides the project default.
- A stage locale. Certain parallel stages allow you to specify that they use a particular locale. This overrides both the project default and the job default.

This manual uses the term *territory* rather than *country* to describe an area that uses a locale.

Time and Date

Most territories have a preferred style for presenting times and dates. For times, this is usually a choice between a 12-hour or 24-hour clock. For dates, there are more variations. Here are some examples of formats used by different locales to express 9.30 at night on the first day of April in 1990:

Territory	Time	Date	InfoSphere DataStage Locale
France	21h30	1.4.90	FR-FRENCH
U.S.	9:30 p.m.	4/1/90	US-ENGLISH
Japan	21:30	90.4.1	JP-JAPANESE

Numeric

This convention defines how numbers are displayed, including:

- The character used as the decimal separator (the radix character)
- The character used as a thousands separator
- Whether leading zeros should be used for numbers 1 through -1

For example, the following numbers can all mean one thousand, depending on the locale you use:

Territory	Number	InfoSphere DataStage Locale
Ireland	1,000	IE-ENGLISH
Netherlands	1.000	NL-DUTCH
France	1 000	FR-FRENCH

Monetary

This convention defines how monetary values are displayed, including:

- The character used as the decimal separator. This can differ from the decimal separator used in numeric formats.
- The character used as a thousands separator. This can differ from the thousands separator used in numeric formats.
- The local currency symbol for the territory, for example, $, £, or ¥.
- The string used as the international currency symbol, for example, USD (US Dollars), NOK (Norwegian Kroner), JPY (Japanese Yen).
- The number of decimal places used in local monetary values.
- The number of decimal places used in international monetary values.
- The sign used to indicate positive monetary values.
- The sign used to indicate negative monetary values.
- The relative positions of the currency symbol and any positive or negative signs in monetary values.

Here are examples of monetary formats different locales use:

Currency	Format	InfoSphere DataStage Locale
U.S. Dollars	$123.45	US-ENGLISH
UK Pounds	£37,000.00	GB-ENGLISH
German Marks	DM123,45	DE-GERMAN
German Euros	€123,45	DE-GERMAN-EURO

Character Type

This convention defines whether a character is alphabetic, numeric, nonprinting, and so on. This convention also defines any casing rules, for example, some letters take an accent in lowercase but not in uppercase.

Collation

This convention defines the order in which characters are collated, that is, sorted. There can be many variations in collation order within a single character set. For example, the character Ä follows A in Germany, but follows Z in Sweden.

Maps and Locales in IBM IBM InfoSphere DataStage Jobs

A large number of maps and locales are installed when you install InfoSphere DataStage with NLS enabled. InfoSphere DataStage makes a distinction between **available** maps and locales and **loaded** maps and locales. Depending on what language you specify when you install InfoSphere DataStage, a set of maps and locales are compiled and loaded ready for use when designing and running InfoSphere DataStage server jobs. Available maps and locales are those that InfoSphere DataStage has available for compiling and loading; these can be specified when designing jobs but must be actually loaded before you run a job that uses them.

You can view what maps and locales are currently loaded and which ones are available from the InfoSphere DataStage Administrator:

1. Open the Administrator client.
2. Click the **Projects** tab to go to the Projects page.
3. Select a project and click the **NLS...** button to open the Project NLS Settings dialog box for that project. By default this shows all the maps currently loaded for server jobs. Choose the **Show all maps** option to see a list of maps available for loading.
4. To view loaded locales click the **Server Locales** tab. Click on the down arrow next to each locale category to see drop down list of loaded locales. Select the **Show all locales** option to have the drop down lists show all the maps available for loading.

Loading Maps

To load one of the available maps so that it can be used by jobs at run time:

1. In the Server Maps page, click the **Install** button. The page expands to show lists of available and loaded maps:
2. Select the map you want to load from the **Available** list on the left and click the **Add** button. A dialog box asks you to confirm the action. Click **Yes**. When the map has been compiled it is added to the **Installed** list on the right. You need to stop and restart the server engine before it is actually loaded, so initially there is no tick beside it.
3. Stop and restart the engine either by rebooting the machine or stopping and starting the IBM InfoSphere DataStage services (see *Administrator Client Guide* for instructions how to do this). The map is then available for jobs at run time.

Loading Locales

To load one of the available locales so that it can be used by jobs at run time:

1. In the Server Locales page, click the **Install** button. The page expands to show lists of available and loaded locales:

2. Select the locale you want to load from the **Available** list on the left and click the **Add** button. A dialog box asks you to confirm the action. Click **Yes**. When the locale has been compiled it is added to the **Installed** list on the right. You need to stop and restart the server engine before it is actually loaded, so initially there is no tick beside it.
3. Stop and restart the server engine either by rebooting the machine or stopping and starting the DataStage services (see *Administrator Client Guide* for instructions how to do this). The locale is then available for jobs at run time.

Using Maps in Server Jobs

You need to use a map whenever you are reading character data (other than 7-bit ASCII) into IBM InfoSphere DataStage or writing character data out of InfoSphere DataStage. The map tells InfoSphere DataStage how to convert the external character set into the internal Unicode character set.

You do **not** need to map data if you are:

- Handling purely numeric data.
- Reading from or writing to a stage representing the internal storage provided by InfoSphere DataStage (that is a Hashed File stage or a UniVerse stage).
- Reading from or writing to an external UniVerse database with NLS enabled.
- Reading or writing 7-bit ASCII data.

InfoSphere DataStage allows you to specify the map to use at various points in a job design:

- You can specify the default map for a project. This is used by all stages in all jobs in a project unless specifically overridden in the job design.
- You can specify the default map for a job. This is used by all stages in a job (replacing the project default) unless overridden in the job design.
- You can specify a map for a particular stage in your job. This overrides both the project default and the job default.
- For certain stages you can specify a map for individual columns, this overrides the project, job, and stage default maps.

Character Data in Server Jobs

You only need to specify a character set map where your job is processing character data. IBM InfoSphere DataStage has a number of character types which can be specified as the SQL type of a column:

- Char
- VarChar
- LongVarChar
- NChar
- NVarChar
- NLongVarChar

All of the above denote string columns, which need to be mapped to the InfoSphere DataStage internal Unicode character set.

Specifying a Project Default Map

You specify the default map for a project in the IBM InfoSphere DataStage Administrator Client:

1. Open the Administrator client.
2. Click the Projects tab to go to the Projects page.

3. Select the project for which you want to set a default map and click the **NLS...** button to open the Project NLS Settings dialog box for that project. By default this shows all the maps currently loaded for server jobs.
4. Choose the map you want from the **Default map name** list. You select the **Show all maps** option and choose a map that is not yet loaded, but note that you will have to load the map (see "Loading Maps") before any jobs that use the map are run.
5. Click **OK**. The selected map is now the default one for that project and is used by all the jobs in that project.

Specifying a Job Default Map

You specify a default map for a particular job in the IBM InfoSphere DataStage Designer using the Job Properties dialog box:

1. Open the job for which you want to set the map in the InfoSphere DataStage Designer.
2. Open the Job Properties dialog box for that job (choose **Edit → Job Properties**).
3. Click the **NLS** tab to go to the NLS page:
4. Choose the map you want from the **Default map for stages** list. You select the **Show all maps** option and choose a map that is not yet loaded, but note that you will have to load the map (see "Loading Maps") before the job is actually run.
5. Click **OK**. The selected map is now the default one for that job and is used by all the stages in that job.

Specifying a Stage Map

You specify a map for a particular stage to use in the stage editor dialog in the IBM InfoSphere DataStage Designer. You can specify maps for all types of stage except:

- Active stages such as the Aggregator and Transformer. These deal with data that has already been input to InfoSphere DataStage and so has already been mapped.
- Stages that use the internal storage offered by InfoSphere DataStage, that is, Hashed File and UniVerse stages. These handle data in the Unicode character set, so require no mapping.

To specify a map for a stage:

1. Open the stage editor in the job in the Designer client. Select the **NLS** tab on the Stage page.
2. Do one of the following:
   - Choose the map you want from the **Map name for use with stage** list. You select the Show all maps option and choose a map that is not yet loaded, but not that you will have to load the map (see “Loading Maps” on page 22) before the job containing this stage is actually run.
   - Click the **Use Job Parameter...** button. This allows you to select an existing job parameter or specify a new one. When the job is run, InfoSphere DataStage will use the value of that parameter for the name of the map to use.
3. Click **OK**. The selected map or job parameter are used by the stage.

Specifying a Column Map

Certain types of server job stage allow you to specify a map that is used for a particular column in the data handled by that stage. The following stages permit per-column mapping:

- ODBC stage
- Sequential File stage

To specify a per-column map:

1. Open the stage editor in the job. Click the **NLS** tab on the Stage page:

2. Select the **Allow per-column mapping** option. Then go to the Inputs or Outputs page (depending on whether you are writing or reading data) and select the **Columns** tab:
3. The columns grid now has an extra field called **NLS Map**. Choose the map you want for a particular column from the drop down list.
4. Click **OK**.

Using Locales in Server Jobs

Locales allows you to specify that data is handled in accordance with the conventions of a certain territory. There is not always a direct relationship between locale and language, for example the French locale is different to the French Canadian one.

Server jobs allow you to choose locales separately for several different aspects of National conventions:

- The format for times and dates
- The format for displaying numbers
- How to display monetary values
- Whether a character is alphabetic, numeric, nonprinting, and so on
- The order in which characters should be sorted (collation)

You can mix locales if required, for example you could specify times and dates in one locale and monetary conventions in another. Descriptions of each type of convention are given in "Locales".

In server jobs you can set a default locale for a project or for an individual job.

Specifying a Project Default Locale

You specify the default locale for a project in the IBM InfoSphere DataStage Administrator Client:

1. Open the Administrator client.
2. Click the **Projects** tab to go to the Projects page.
3. Select the project for which you want to set a default map and click the **NLS...** button to open the Project NLS Settings dialog box for that project. Click the **Server Locales** tab to go to the Server Locales page.
4. Click on the arrow next to the category for which you want to set a locale, and choose a locale from the drown down list. You can select the **Show all locales** option and choose a locale that is not yet loaded, but note that you will have to load the locale (see "Loading Locales") before you run jobs that use it.
5. Click **OK**. The selected locale is now the default one for that category in the project and is used by all the jobs in that project.

Specifying a Job Default Locale

You specify a default locale for a particular job in the IBM InfoSphere DataStage Designer, using the Job Properties dialog:

1. Open the job for which you want to set the locale in the Designer client.
2. Open the Job Properties dialog box for that job (choose **Edit → Job Properties**).
3. Click the **NLS** tab to go to the NLS page:
4. Click on the arrow next to the category for which you want to set a locale, and choose a locale from the drown down list. You can select the **Show all locales** option and choose a locale that is not yet loaded, but note that you will have to load the locale (see "Loading Locales") before the job is actually run.

5. Click **OK**. The selected locale is now the default one for that category in the job and is used by all the stages in that job.

Chapter 4. Server job stages

Complex Flat File Stages

The Complex Flat File stage lets you convert data extracted from complex flat files that are generated on an IBM mainframe. A complex flat file has hierarchical structure in its arrangement of columns. It is physically flat (that is, it has no pointers or other complicated infrastructure), but logically represents parent-child relationships. You can use multiple record types to achieve this hierarchical structure.

Recognizing a Hierarchical Structure

For example, use records with various structures for different types of information, such as an 'E' record for employee static information, and a 'S' record for employee monthly payroll information, or for repeating groups of information (twelve months of revenue). You can also combine these record groupings, and in the case of repeating data, you can flatten nested OCCURS groups.

Managing Repeating Groups and Internal Structures

You can easily load, manage, and use repeating groups and internal record structures such as GROUP fields and OCCURS. You can ignore GROUP data columns that are displayed as raw data and have no logical use for most applications. The metadata can be flattened into a normalized set of columns at load time, so that no arrays exist at run time.

Selecting subsets of columns

You can select a subset of columns from a large COBOL File Description (CFD). This filtering process results in performance gains since the stage no longer parses and processes hundreds of columns if you only need a few.

Complex flat files can also include legacy data types.

Output Links

The Complex Flat File stage supports multiple outputs. An output link specifies the data you are extracting, which is a stream of rows to be read.

When using the Complex Flat File stage to process a large number of columns, for example, more than 300, use only one output link in your job. This dramatically improves the performance of the GUI when loading, saving, or building these columns. Having more than one output link causes a save or load sequence each time you change tabs.

The Complex Flat File stage does not support reference lookup capability or input links. For information about jobs built with Version 1 of this stage, see the next section.

Existing Jobs Built with Version 1 of the Complex Flat File Stage

When you first open the Complex Flat File stage, the grid for the **Source Columns** tab is empty for existing jobs until you click **OK** and save the job. The GUI generates the source column values using the output columns list, and you can upgrade existing jobs by selecting **Yes** to the following query:

```
This stage was developed with a previous version of the CFF plugin. Would you like to upgrade this stage?
```

The source columns are populated, and you can select and modify columns.

You can stop the upgrade by selecting **No** to the query. You cannot use the **Source Columns** tab, and grid remains empty. Existing jobs run unaltered as long as you do not save the job.

Functionality

Supported Functionality

The Complex Flat File stage supports the following functionality:

- EBCDIC and ASCII raw data.
- The following COBOL data types:
  - Packed (COMP-3).
  - Zoned (signed DISPLAY).
  - Character (DISPLAY or PIC X).
  - Integer (COMP. Integer also supports nonstandard 1- and 3 byte implementations).
  - Binary (same as COMP, but unsigned.)
  - Decimal (COMP-2).
  - Float (COMP-1).
  - Fixed format structures. It does not support columns that are separated by delimiters.
- One REDEFINES clause.
- Parallel OCCURS by using separate output links.
- Nested OCCURS in the same COBOL File Description (CFD) format. They must be flattened at load time (not at run time).
- Fixed OCCURS with no DEPENDING ON clause.
- Complex flat files generated on an IBM mainframe.
- NLS (National Language Support).
- The ability to select subsets of columns from a CFD.

Unsupported Functionality

The following functionality is not supported:

- Non 8 bit byte file formats.
- Complex flat files generated on non-IBM mainframes (for example, Windows).

Terminology

The following list describes terms used in this document:

Term **Description**

CFD COBOL File Description format.

Complex flat file
A superset of the simple flat file, but it contains complex data structures. The complex data structures supported by Complex Flat File are groups, arrays, and redefines, which are found in VSAM or QSAM files. (Load-ready or delimited files do not contain these complex structures.) A complex flat file has hierarchical structure implied in its arrangement of columns. Complex flat files can also include legacy data types.

DCLGen Import
The IBM InfoSphere DataStage component to import a table definition in a DCLGen file that was previously exported from VS/IBM DB2®.

Fixed-width flat file
A file characterized by delimited or fixed length (binary) files.

Flattening
The conversion of files containing complex data structures, such as arrays, groups, and redefines, into data files that contain records with no structured relationships.

Normalization
The conversion of records in NF^2 (nonfirstnormal form) format, containing multi-valued data, into one or more 1NF (first normal form) rows.

QSAM
Queued Sequential Access Method.

VSAM
Virtual Storage Access Method. This method is a file management system for the IBM MVS™ mainframe operating system.

Using the Complex Flat File Stage

When you use the custom GUI to edit a Complex Flat File stage, the **CFF** Stage dialog box opens. This dialog box has the **Stage** and **Output** pages:

- **Stage.** This page displays the name of the stage you are editing in the **Stage name** field. The **General** tab describes the purpose of the stage in the **Description** field. (The **NLS** tab appears only if you have installed NLS. For details, see "NLS Tab.")

  Note: You cannot change the name of the stage from this dialog box.
- **Output.** This page specifies the data sources to use and the associated column definitions for each output link.

NLS Tab

You can define a character set map that interprets the input file for a stage. Do this on the **NLS** tab on the Stage page, which appears only if you have installed NLS.

If NLS is installed, the **NLS** option is selected in the **Data Format** list box on the **General** tab of the Output page, and the **NLS** tab on the Stage page is enabled.

Specify information using the following button and fields:

- **Map name to use with stage.** The default character set map is defined for the project or the job. You can change the map by selecting a map name from the list.
- **Use Job Parameter ...**. Specifies parameter values for the job. Use the format #*Param*#, where *Param* is the name of the job parameter. The string #*Param*# is replaced by the job parameter when the job is run.
- **Show all maps.** Lists all the maps that are shipped with IBM InfoSphere DataStage.
- **Loaded maps only.** Lists only the maps that are currently loaded.

If NLS is not installed, the **NLS** option is unavailable in the **Data Format** list box and the **NLS** tab on the Output page is disabled. The input file is interpreted according to the option that is selected in the **Data Format** list box on the **General** tab of the Output page.

Defining an Output Link

When you read data from a data source, a Complex Flat File stage has an output link.

Define the properties of this link and the column definitions of the data on the Output page in the CFF Stage dialog box.

About the Output Page

The Output page has an **Output name** field, the **General**, **Source Columns, Select Columns**, **Selection Criteria,** and**Destination Columns** tabs, and the **Columns...** and **View Data...** buttons. (The **NLS** tab appears only if you have installed NLS. For details, see "NLS Tab" on page 29.)

- **Output name.** The name of the output link. Select the link you want to edit from the **Output name** list box. This list displays all the output links from the Complex Flat File stage.
- **Columns... button.** Click the **Columns...** button to display a brief list of the columns in the data source file. As you enter detailed metadata on the **Source Columns** tab, you can leave this list displayed.
- **View Data... button.** Click the **View Data...** button to start the Data Browser. This lets you look at the data associated with the output link.

General Tab

This tab is displayed by default. You can optionally enter text to describe the purpose of the output link in the **Description** field. Enter the appropriate information for the following fields:

- **Path.** The input path name of the data source to retrieve data from. You can also click the ... button at the right of the field to browse the directories on the computer hosting the engine tier for the data source.
- **Data Format.** The format of the input file: EBCDIC or ASCII, or NLS. If NLS is enabled, the Data Format is set to NLS and is not editable. (NLS maps support EBCDIC data.)
- **Record Style.** The end-of-line treatment for records according to the following table:

Table 2. End-of-line treatment

Data Format	Record Style	Record Length	Comments
ASCII or EBCDIC	Binary	Greater than zero	The record length is determined by the **Record Length** field. If the metadata defines a record that is longer than the **Record Length** field, the columns that have start positions greater than **Record Length** are set to Null.
ASCII or EBCDIC	Binary	Zero	The record length is determined by the metadata.

Table 2. End-of-line treatment (continued)

Data Format	Record Style	Record Length	Comments
ASCII or EBCDIC	CR/LF	Any value	The record length is determined by the CR/LF delimiter. If the metadata defines a record that is longer than **Record Length** as determined by the CR/LF, the columns that have start positions greater than the position of the CR/LF are set to NULL. If the metadata defines a record that is less than the position of the CR/LF, the data after the end of the record and before the CR/LF (as determined by the metadata) is discarded. If the data being read is in EBCDIC format, coincidental CR/LF characters can occur. These also delimit a record.
NLS	Binary	> 0	The record length is determined by the **Record Length** field. If the metadata defines a record that is longer than the **Record Length** field, the columns that have start positions greater than **Record Length** are set to spaces.

Table 2. End-of-line treatment (continued)

Data Format	Record Style	Record Length	Comments
NLS	CR/LF	Any value	The record length is determined by the CR/LF delimiter. If the metadata defines a record that is longer than **Record Length** as determined by the CR/LF, the columns that have start positions greater than the position of the CR/LF are set to spaces If the metadata defines a record that is less than the position of the CR/LF, the data after the end of the record and before the CR/LF (as determined by the metadata) is discarded. If the data being read is in EBCDIC format, coincidental CR/LF characters can occur. These also delimit a record.
NLS	Binary	0	The record length is determined by the metadata.

Note: If **Record Style** is set to CR/LF, the last record in the data set should not be terminated by CR/LF. If one or more CR/LFs are at the end of the data set, empty records are generated for each CR/LF.

- **Verify sign value in DECIMAL COMP-3 data**. If selected, the stage checks for a valid sign value in data defined as COMP-3. If the value is other than a hexadecimal "C," "F," or "D," the stage writes an error message to the IBM InfoSphere DataStage log.
- **Preserve NULL.** If selected, the stage interprets the value of any column containing binary null values as a SQL_NULL. If not selected, the stage interprets the value as zero. The default is **Preserve NULL** not selected.
- **Description**. Optional text describing the purpose of the output link.

Source Columns Tab

The **Source Columns** tab contains the full list of columns. Using the **Select Columns** tab, you can select columns from this list to output on the CFF output link. (You can also use the Transformer stage to do this, but slower performance results because of the unused column data.)

Use the **Source Columns** tab to enter the column data that make up the flat file manually, or you can use **Load**.

Note: Do not use the **Source Columns** tab to omit columns from the destination file. **Source Columns** must accurately and completely describe the source file, or the file will not be parsed correctly. Rather, use the Select Columns tab to omit columns.

Load Button

Load automatically flattens the arrays by generating column names with monotonically increasing numbers if you answer **Yes** to the following query:

```
Do you want to flatten the occurs in the columns being loaded?
```

If you enter data manually, you should flatten any arrays manually.

The following columns have special meaning for the Complex Flat File stage:

- **Level number.** Represents the level number of the column within a COBOL file description.
- **Native type.** Represents the COBOL data type.

The other columns in the grid are the standard columns for the **Columns** tab. You can edit these fields by right-clicking on a row in the grid and selecting **Edit row...** . This opens the Edit Column dialog box.

Edit Column Meta Data Dialog Box

Use the Edit Column Meta Data dialog box to edit the column metadata as described in this section.

The Edit Column Meta Data dialog box has a general area containing fields that are common to all data source types, plus two pages containing fields specific to metadata used in server jobs and information specific to COBOL data sources.

Meta Data Properties

Use the IBM InfoSphere DataStage and QualityStage Designer to import the complex flat file metadata (CFD) into the InfoSphere DataStage. The following list defines the properties that are captured for complex flat file metadata.

Field Description

Column name
The name of the column.

Key Specifies whether this column is a key.

Native type
The native data type of the column.

Use one of the following values: FLOAT, DECIMAL, BINARY, DISPLAY_NUMERIC, CHARACTER, or GROUP.

For details about using the GROUP native type to handle dates, see “Destination Columns Tab” on page 37.

Length
The numeric value of the precision.

Scale The number of decimal places.

Nullable
Specifies whether the column can contain null values. If set to Yes, it is subject to a NOT NULL constraint. (Yes/No)

Date format
The format of a date column.

Description
The description of the column. See examples of **Description** field values at the end of this section.

Level number
The relative COBOL level number of the column.

Occurs
The number of occurrences of the column specified in the COBOL OCCURS clause. (See examples of parallel OCCURS, which are unsupported, at the end of this section.)

Usage Specifies a COBOL usage clause. Use one of the following:

COMP COMP-1 COMP-2 COMP-3 DISPLAY

Sign indicator
Specifies the sign. Set it to S if the picture character string contains the S symbol. Otherwise, set it to U.

Sign option
If you specify a sign clause and the picture character-string contains an S symbol, this attribute is set to one of the following values:

L - LEADING T - TRAILING LS - LEADING SEPARATE TS - TRAILING SEPARATE

Sync indicator
Specifies whether this is a COBOL synchronized clause.

Redefined field
The name of the column being redefined.

Depending on
The name of the depending column.

Storage length
The actual storage length in bytes of the column.

Picture
Displays a generated Picture clause based on the value in **Native type**, **Length**, and **Scale**.

Processing the Metadata

Use the buttons at the bottom of the Edit Column Meta Data dialog box to continue adding or editing columns, or to save the changes and close the dialog box.

- **Previous** and **Next.** View the metadata in the previous or next row.
- **Close.** Close the Edit Column Meta Data dialog box. If there are outstanding changes to the current row, you are asked whether you want to save them before closing.
- **Apply.** Save changes to the current row.
- **Reset.** Remove all changes made to the row since the last time you applied changes.

Only a subset of these properties is visible on the **Source Columns** tab. To see all the properties for a given row, right-click on a row in the grid, and select **Edit row...**.

If you enter or modify metadata using the stage editor, and you want to save a copy in the InfoSphere DataStage repository for use in another stage, click the **Save...** button.

To load an existing table definition into the stage, use the **Load...** button.

Handling Parallel OCCURS

The Complex Flat File stage does not support parallel OCCURS, that is, two or more OCCURS clauses in the same data definition. You need to process these parallel OCCURS clauses down separate output links.

This example uses a PHONES OCCURS clause and an ADDRESS OCCURS clause:

```
01 CLIENT.
   03 SURNAME            PIC X(25).
   03 FORENAME           PIC X(25).
```

```
   03 ADDRESS            OCCURS 4.
      05 ADDLINE         PIC X (10).
   03 POSTCODE           PIC X (10).
   03 PHONES             OCCURS 2.
      05 TELNO           PIC X(10).
```

The next example uses a PHONES OCCURS clause and an ADDLINE OCCURS clause:

```
01 CLIENT.
   03 SURNAME            PIC X(25).
   03 FORENAME           PIC X(25).
   03 ADDRESS.
      05 ADDLINE         PIC X (10) OCCURS 5.
   03 POSTCODE           PIC X (10).
   03 PHONES             OCCURS 2.
      05 TELNO           PIC X(10).
```

The CFF custom GUI recognizes the parallel OCCURS and displays the following error:

```
Too many occurs
```

You are not allowed to save the loaded or edited column definitions.

To process parallel OCCURS:

1. Clear the **Occurs** field using the Edit Column Meta Data dialog box.
2. Enter NONE in the **Description** field of the columns that are not being processed on this link. This lets the data for those columns flow through unchanged.
3. Create a separate output link using a similar procedure to process the next OCCURS.

Description Field Values

The following values for the **Description** field have special meaning at run time:

- **UNSIGNED_DECIMAL.** Use only with DECIMAL fields. You can use this value with packed decimal fields to trigger special unpacking algorithms.
- **ANYSIGN_DECIMAL.** Use only with DECIMAL fields. You can use this value with packed decimal fields to trigger special unpacking algorithms.
- **NONE.** Use with non-GROUP native types. NONE causes the data to flow through the stage unchanged, that is, no conversions are done on the data, and raw data is output. NONE is ignored if you use a date format.
- **OCCURS_COUNTER.** Behaves as a pseudo-column of the DECIMAL native type that does not expect any data in the input stream.

You must first insert a new field with the DECIMAL type into the **Columns** grid within your OCCURS clause. You must also include the OCCURS_COUNTER string in the **Description** field. At run time the stage creates its own data by automatically increasing the counter for each occurrence that is processed. This is not currently supported for nested OCCURS.

GROUP Columns and OCCURS

If a column in a GROUP that has an OCCURS is selected, and the GROUP column is not selected, incorrect results might be displayed. You should include the GROUP column with the OCCURS in your selection if any column in the GROUP is selected.

Note: Only certain types of GROUP columns might be selected. See the following sections for details.

Select Columns Tab

Use the **Select Columns** tab to choose which columns to load on the output link. The **Select Columns** tab contains a grid displaying the column definitions for the data being output on the chosen link.

Note: Do not use the **Source Columns** tab for the purpose of selecting columns for the destination file. Use only the**Select Columns** tab for this purpose. The description on the Source Columns tab must match the source file completely and accurately (see "Source Columns Tab" on page 32).

The **Select Columns** tab functions similarly to that of the Complex Flat File mainframe source stage, as follows:

- **Available columns** lists the source columns displayed in hierarchical format. It uses fields for non-GROUP columns and folders for GROUP columns. As you select or clear each column, a check mark appears on the column in the list.

  You can only select GROUP columns if all the columns in the GROUP have the CHARACTER data type. If any column in the GROUP has a different data type, you cannot select the GROUP column and is displayed as such.
- **Selected columns** contains the list of columns you create using the arrow keys.
- Use these arrow keys to move columns back and forth between the **Available columns** list and the **Selected columns** list. Use the single arrow (>) to move highlighted columns, the double arrow (>>) to move all items.
- By default all columns are selected for loading. Click **Find** to open a dialog box which lets you search for a particular column.
- Click **OK** when your selection is complete to load the selected columns.

Selection Criteria Tab

Use the **Selection Criteria** tab to redefine fields extracted from the input file. Enter the appropriate information for the following fields:

- **Start Record #.** The record number at which to start processing.
- **End Record #.** The record number at which to stop processing.
- **ID Field.** Choose the field containing the record type value from the list box.
- **Value (Hex).** The value of the record type. This value is converted to the **Record Style** before comparison, for example, ASCII or EBCDIC. Only the records that contain this value are sent to the output link. Value ranges are unsupported. If the value is preceded by the ampersand (&) character, it is treated as a hexadecimal value and compared without any conversion.

Redefined Example

The Complex Flat File stage supports the redefinition of any portion of the source file. It does this by resetting the start position of a field that has its metadata redefine another field.

For example, Field-2 redefines Field-1, and so forth.

Input:

```
01 Example-Record.
   03 Field-1                Pic X(24).
   03 Field-2 redefines Field-1.
      05 Field-2a            Pic X(8).
      05 Field-2b            Pic X(8).
      05 Field-2c            Pic X(8).
   03 Field-3                Pic X(24).
   03 Field-3 redefines Field-3.
      05 Field-4a            Pic X(8).
      05 Field-4b            Pic X(8).
      05 Field-4c            Pic X(8).
   03 Field-5 redefines Field-1.
```

```
      05 Field-5a             Pic X(8).
      05 Field-5b             Pic X(8).
      05 Field-5c             Pic X(8).
Input Data:
2a2a2a2a2b2b2b2b2c2c2c2c4a4a4a4a4b4b4b4b4c4c4c4c
Output Field Order:
Field-2a
Field-2b
Field-2c
Field-4a
Field-4b
Field-4c
Field-5a
Field-5b
Field-5c
Output Data:
2a2a2a2a2b2b2b2b2c2c2c2c4a4a4a4a4b4b4b4b4c4c4c4c2a2a2a2a2b2b2b2b2c2c2c2c
```

Destination Columns Tab

The **Destination Columns** tab on the Output page contains the list of columns that you created using the **Select Columns** tab. The columns are grayed out and are not editable. You must use the **Select Columns** tab to choose which columns to load on the output link.

Date Considerations

In many cases, COBOL files define dates as a character field. For example:

```
05 Application-Date     pic 99999999.
```

Click the **Source Columns** tab on the Output page to enter or load column definitions for your data. In this case, define the **Native type** field for Application-Date as CHARACTER. Select an appropriate format from the **Date format** field of the Edit Column Meta Data window, in this case, CCYYMMDD.

To generate an IBM InfoSphere DataStage date in the column in the output link, the input data and the **Date format** field must use the same format. For example, the input data in the format "25/12/2000" must use the DD/MM/CCYY format in the **Date format** field. Otherwise, a date with a null value is generated, and a warning about a bad date conversion appears in the InfoSphere DataStage log.

Folder Stages

Folder stages are used to read or write data as files in a directory located on the IBM InfoSphere DataStage server.

Using Folder Stages

Folder stages can read multiple files from a single directory and can deliver the files to the job as rows on an output link. Folder stages can also write rows of data as files to a directory. The rows arrive at the stage on an input link.

Note: The behavior of the Folder stage when reading folders that contain other folders is undefined.

In an NLS environment, the user running the job must have write permission on the folder so that the NLS map information can be set up correctly.

When you edit a Folder stage, the **Folder Stage** dialog box appears. This dialog box has up to three pages:

- **Stage**. The **General** tab displays the name of the stage you are editing, the stage type and a description. The **Properties** tab contains properties which define the operation of the stage.

- **Inputs**. The **Columns** tab displays the column definitions for data arriving on the input link. The directory to which the stage writes the files is defined in the **Folder pathname** property on the Stage **Properties** tab.
- **Outputs**. The **Columns** tab displays the column definitions for data leaving on the output link. The **Properties** tab controls the operation of the link. The directory from which the stage reads the files are defined in the **Folder pathname** property on the Stage **Properties** tab.

Defining Character Set Maps

You can define a character set map for a Folder stage by using the **NLS** tab in the Folder Stage dialog box.

The default character set map (defined for the project or the job) can be changed by selecting a map name from the list. The tab also has the following fields:

- **Show all maps**. Lists all the maps supplied with IBM InfoSphere DataStage. Maps cannot be used unless they have been loaded by using the Administrator client.
- **Loaded maps only**. Displays the maps that are loaded and ready for use.
- **Use Job Parameter...** . Allows you to specify a character set map as a parameter to the job containing the stage. If the parameter has not yet been defined, you are prompted to define it from the Job Properties dialog box.

Defining Folder Stage Input Data

The Folder stage only has input data when it is being used to write files to a directory. In this case the directory being written to is defined on the **Properties** tab on the Stage page.

The Inputs page **Properties** tab defines properties for the input link. The properties are as follows:

- **Preserve CRLF**. When **Preserve CRLF** is set to **Yes**, field marks are not converted to newlines on write. It is set to **Yes** by default.

The **Columns** tab defines the data arriving on the link to be written in files to the directory. The first column on the **Columns** tab must be defined as a key, and gives the name of the file. The remaining columns are written to the named file, each column separated by a newline. Data to be written to a directory would normally be delivered in a single column. For example, the columns grid might look like this:

Column name	Key	SQL type	Length	Scale	Nullable
FileName	✓	VarChar	255		No
Record		LongVarChar	999999		No

Defining Folder Stage Output Data

The behavior of the output link is controlled by the output properties on the Outputs **Properties** tab.

The properties are as follows:

- **Sort order**. Choose from **Ascending**, **Descending**, or **None**. This specifies the order in which the files are read from the directory.
- **Wildcard**. This allows for simple wildcarding of the names of the files found in the directory. Any occurrence of * (asterisk) or ... (three periods) is treated as an instruction to match any or no characters.
- **Preserve CRLF**. When **Preserve CRLF** is set to **Yes**, newlines are not converted to field marks on read. It is set to **Yes** by default.

- **Fully qualified**. Set this to yes to have the full path name of each file written in the key column instead of just the file name.

The **Columns** tab defines a maximum of two columns. The first column must be marked as the key and receives the file name. The second column, if present, receives the contents of the file. You can load these column definitions from the default table definition that is supplied for the stage. Click **Load** and select the Folder table definition located in the `Table Definitions\Built-in\Examples` folder in the repository tree.

Hashed File Stages

Hashed File stages represent a hashed file, that is, a file that uses a hashing algorithm for distributing records in one or more groups on disk. Use a Hashed File stage to access UniVerse files. The server engine can host UniVerse files locally. You can use a hashed file as an intermediate file in a job, taking advantage of DSEngine's local hosting.

See *IBM InfoSphere DataStage and QualityStage Connectivity Guide for IBM UniVerse and UniData* for more information about using Hashed File stages to access UniVerse files.

Using a Hashed File Stage

You can use a Hashed File stage to extract or write data, or to act as an intermediate file in a job. The primary role of a Hashed File stage is as a reference table based on a single key field.

Each Hashed File stage can have any number of inputs or outputs. When you edit a Hashed File stage, the **Hashed File Stage** dialog box appears. This dialog box can have up to three pages (depending on whether there are inputs to and outputs from the stage):

- **Stage**. Displays the name of the stage you are editing. This page has a **General** tab, where you can enter text to describe the purpose of the stage in the **Description** field and specify where the data files are by clicking one of the option buttons:
  - **Use account name**. If you choose this option, you must choose the name of the account from the **Account name** list. This list contains all the accounts defined in the **Table Definitions → Hashed** folder in the repository. If the account you want is not listed, you need to define a table definition. Alternatively, you can enter an account name or use a job parameter. For details about how to create a table definition, or how to define and use job parameters, see *IBM InfoSphere DataStage and QualityStage Designer Client Guide*.
  - **Use directory path**. If you choose this option, you must specify a directory path containing the UV account. The directory must be a UniVerse account and is used for UniVerse accounts that do not appear in the UV.ACCOUNT file. If the hashed file is hosted locally by DSEngine, you need to specify the IBM InfoSphere DataStage project directory as the directory path, for example, C:\IBM\InformationServer\Server\Projects\Dstage. The directory is specified in the **Directory path** field. You can enter a path directly, click **Browse...** to search the system for a suitable directory, or use a job parameter.
  - **SQL NULL value**. Determines what character represents the SQL null value in the hashed file corresponding to this stage. If your system will be using the Euro symbol, select the **Special (allow Euro)** option from the list. Select **Auto detect** to have InfoSphere DataStage determine what represents SQL null.
  - **UniVerse Stage Compatibility**. Select this check box to ensure that any job conversions will work correctly. With this option selected, the date or time will be represented in ISO format (depending on the Extended type) and numerics will be scaled according to the metadata. (The job conversion utility is a special standalone tool - it is not available in the Designer client.)
- **Inputs**. This page is only displayed if you have an input link to this stage. Specifies the data file to use and the associated column definitions for each data input link. This page also specifies how data is written to the data file.

- **Outputs**. This page is displayed only if you have an output link to this stage. Specifies the data file to use and the associated column definitions for each data output link.

Click **OK** to close this dialog box. Changes are saved when you save the job.

If a Hashed File stage references a hashed file that does not already exist, use the Director Validate Job feature before you run the job, and InfoSphere DataStage will create it for you. To validate a job, choose **Job → Validate** from the Director client. The Job Run Options dialog box appears. Click **Validate**. For more information about validating a job and setting job options, see *IBM InfoSphere DataStage and QualityStage Director Client Guide*.

Defining Hashed File Input Data

When you write data to a hashed file, the Hashed File stage has an input link. The properties of this link and the column definitions of the data are defined on the Inputs page in theHashed File Stage dialog box.

The Inputs page has the following field and two tabs:

- **Input name**. The name of the input link. Choose the link you want to edit from the **Input name** list. This list displays all the input links to the Hashed File stage.
- **General**. Displayed by default. Contains the following fields and options:
  - **File name**. The name of the file the data is written to. You can either use a job parameter to represent the file created during run time or choose the file from the **File name** list. This list contains all the files defined in the **Table Definitions → Hashed▸*Account name*** folder in the repository, where *Account name* is the name of the account chosen on the Stage page. By default the name of the input link is used as the file name. If the file you want is not listed, you need to define a table definition.
  - **Clear file before writing**. If you select this check box, the existing file is cleared and new data records are written to the empty file. This check box is cleared by default.
  - **Backup existing file**. If you select this check box, a backup copy of the existing file is made before the new data records are written to the file. The backup can be used to reset the file if a job is stopped or aborted at run time. This check box is cleared by default.
  - **Allow stage write cache**. Select this check box to specify that all records should be cached, rather than written to the hashed file immediately. Avoid this when your job writes and reads to the same hashed file in the same stream of execution, for example, where a Transformer stage checks if a record already exists to determine the required operation. (If you have caching on the server enabled, any caching attributes that the file was created with will override the stage-level caching).
  - **Create File**. Select this check box to specify that the stage will create the hashed file for writing to. Click **Options** to open the Create file options dialog box to specify details about how the file is created (see "Create File Options" on page 41).
  - **Description**. Contains an optional description of the input link.
- **Columns**. Contains the column definitions for the data written to the file.

  Note: You should use the **Key** check boxes to identify the key columns. If you don't, the first column definition is taken as the hashed file's key field. The remaining columns dictate the order in which data will be written to the hashed file. Do *not* reorder the column definitions in the grid unless you are certain you understand the consequences of your action.

Click **View Data...** to open the Data Browser. This enables you to look at the data associated with the input link. For a description of the Data Browser, see *IBM InfoSphere DataStage and QualityStage Designer Client Guide*.

Create File Options

If you choose to create the hashed file to write to, the Create file options dialog box allows you to specify various options about how the file is created.

The dialog box contains the following fields:

- **File type**. The file type chosen determines what other options are available in the dialog box. The default is **Type30(Dynamic)**.
- **Minimum modulus**. Visible only for Type30(Dynamic) file types. Specifies the dynamic file minimum modulus in the range 1 to 999999. The default is 1.
- **Group size**. Visible only for Type30(Dynamic) file types. Specifies the dynamic group size. Choose 1 to select a group size of 2048 bytes, or 2 to select a group size of 4096 bytes. The default is 1.
- **Split load**. Visible only for Type30(Dynamic) file types. Specifies the dynamic file split as a percentage in the range 1 to 99. The default is 80.
- **Merge load**. Visible only for Type30(Dynamic) file types. Specifies the dynamic file merge load as a percentage in the range 1 to 99. The default is 50.
- **Large record**. Visible only for Type30(Dynamic) file types. Specifies the large record value in bytes in the range 1 to 999999. The default is 1628.
- **Hash algorithm**. Visible only for Type30(Dynamic) file types. Specifies the dynamic file hashing algorithm. Choose from **GENERAL** or **SEQ.NUM**. The default is **GENERAL**.
- **Record size**. Visible only for Type30(Dynamic) file types. Specifies the record size in the range 1 to 999999.
- **Modulus**. Visible only for hashed file types. Specifies the hashed file modulus in the range 1 to 999999. The default is 1.
- **Separation**. Visible only for hashed file types. Specifies the hashed file separation in the range 1 to 999999. The default is 2.
- **Caching attributes**. If you have server caching enabled, this allows you to choose caching attributes for the file you are creating. These attributes will stay with the file wherever it is used subsequently. **NONE** means no caching is performed, **WRITE DEFERRED** is the fastest method, but file integrity can be lost if a system crash should occur. **WRITE IMMEDIATE** is slower, but safer in file integrity terms.
- **Minimize space**. Visible only for Type30(Dynamic) file types. Select this to specify that some of the other options are adjusted to optimize for minimum file size.
- **Delete file before create**. Select this check box to specify that any existing file of the same name is deleted before a new one is created.

Defining Hashed File Output Data

When you extract data from a hashed file, the Hashed File stage has an output link. The properties of this link and the column definitions of the data are defined on the Outputs page in the Hashed File Stage dialog box.

The Outputs page has the following two fields and three tabs:

- **Output name**. The name of the output link. Choose the link you want to edit from the **Output name** list. This list displays all the output links from the Hashed File stage.
- **Normalize on**. This list allows you to normalize (or unnest) data. You can normalize either on an association or on a single unassociated multivalued column. The **Normalize on** list is only enabled for nonreference output links where metadata has been defined that contains multivalued fields.
- **General**. Displayed by default. Contains the following fields and options:
  - **File name**. The name of the file the data is read from. You can use a job parameter to represent the file created during run time or choose the file from the **File name** list. This list contains all the files

defined in the **Table Definitions** → **Hashed**▸*Account name* folder in the repository, where *Account name* is the name of the account chosen on the Stage page. If the file you want is not listed, you need to define a table definition.

- **Record level read**. Select this to force the file to be read record by record. This is slower, but is necessary if you want to read and write the hashed file at the same time. If you specify a select statement on the **Selection** tab, the file is read at the record level anyway and this check box is selected but grayed out.
- **Pre-load file to memory**. You can use these options to improve performance if the output link is a reference input to a Transformer stage. If you select **Enabled**, the hashed file is read into memory when the job is run (**Disabled** is selected by default). The remaining two options are for specialist use and cater for situations where you need to modify a lookup table as a job runs. If **Enabled, Lock for Updates** is selected the hashed file is read into memory when the job is run. If a lookup is not found in memory, the job looks in the file on disk. If the lookup is still not found, an update lock is taken in the knowledge that the record will subsequently be written in the hashed file by this job. The operation for **Disabled, Lock for Updates** is similar, except that the hashed file is not read into memory.
- **Description**. Contains an optional description of the output link.

- **Columns**. Contains the column definitions for the data on the chosen output link.

  You should be aware of the following issues when outputting data from a hashed file:

  - Key fields should be identified by selecting the **Key** boxes. (If you fail to do this be warned that the first column will be treated as the key, which might lead to undesired results).
  - By default other columns are ordered according to their position in the file. You can also use the hashed file stage to reorder columns as they are read in. Do this by specifying the column order in the **Position** field. The columns will then be written to the output link in that order, although they retain the same column names. If you use this feature you should identify the key column or columns by setting their **Position** field to 0.
  - Do not reorder the column definitions in the grid unless you are certain you understand the consequences of your action. You should be especially wary of using the **Position** field to reorder columns and then saving the definition as a table definition in the repository for subsequent reuse. In particular, if you use this column definition to write to the same hashed file, you will be reordering the file itself.
  - You can output an entire record as a single column if required. Do this by inserting a value of -1 in the **Position** field of the record field's column definition. (The key column **Position** field should be 0.)
- **Selection**. Contains optional SELECT clauses for the conditional extraction of data from a file. This tab is only available if you have specified the hashed file by account name, rather than directory path, on the Stage page.

Click **View Data...** to open the Data Browser. This enables you to look at the data associated with the output link.

If you intend to read and write from a hashed file at the same time, you must either set up a selection on the **Selection** tab, or you should select the **Record level read** check box on the **General** tab. This ensures the file is read in records rather than in groups, and that record locks can operate. Note, however, that this mode of operation is much slower and should only be used when there is a clear need to read and write the same file at the same time.

Using the Euro Symbol on Non-NLS systems

If you want to include the Euro symbol in hashed files on non-NLS systems, you have to take some steps to support the symbol. The steps you take depend on what type of system you are running your IBM InfoSphere DataStage server on.

UNIX Systems using ISO 8859-15 code page

To support the Euro symbol on this system you need to edit the file *msg.txt* in the IBM InfoSphere DataStage home directory as follows:

- In line LOC0016 replace the $ with the Euro symbol (you can generate a Euro symbol using a keyboard that generates it, or use the BASIC command char(128) or char(164).
- In line LOC0015 ensure the proper decimal separator is set.
- In line LOC0014 ensure the proper thousand separator is set.

Windows Systems and UNIX Systems using the Windows Code Page

On these systems the code that represents the Euro system can clash with the hashed files representation of SQL null. A number of steps are needed to overcome this problem:

- If your system will never require a Euro symbol to appear in isolation in a column of a hashed file, then all you need do is edit the file *msg.txt* in the IBM InfoSphere DataStage home directory as follows:
  - In line LOC0016 replace the $ with the Euro symbol (you can generate a Euro symbol using a keyboard that generates it, or use the BASIC command char(128) or char(164).
  - In line LOC0015 ensure the proper decimal separator is set.
  - In line LOC0014 ensure the proper thousand separator is set.
- If your system does require to use a Euro symbol in isolation, then you need to choose another character to represent SQL null. This is done on the **General** tab on the Hashed File Stage page. Choose one of the following options from the **SQL NULL value** list:
  - **Special (allow Euro)**. This sets SQL null to 0x19.
  - **Auto Detect**. Detects if Euro is the local currency symbol and, if it is, sets SQL null to 0x19.

Sequential File Stages

Sequential File stages are used to extract data from, or write data to, a text file. The text file can be created or exist on any drive that is either local or mapped to the server. Each Sequential File stage can have any number of inputs or outputs.

Using a Sequential File Stage

When you edit a Sequential File stage, the Sequential File Stage dialog box appears. This dialog box can have up to three pages (depending on whether there are inputs to and outputs from the stage):

- **Stage**. Displays the name of the stage you are editing. The **General** tab also allows you to specify line termination options, an optional description of the stage, and whether the stage uses named pipes or filter commands.

  The line termination options let you set the type of line terminator to use in the Sequential File stage. By default, line termination matches the type used on your IBM InfoSphere DataStage server. To change the value, choose one of **Unix style (LF)**, **DOS style (CR LF)**, or **None**.

  Select the **Stage uses named pipes** check box if you want to use the named pipe facilities. These allow you to split up a large job into a number of smaller jobs. You might want to do this where there is a large degree of parallelism in your design, as it will increase performance and allow several developers to work on the design at the same time. With this check box selected, all inputs and outputs to the stage use named pipes. Examples of how to use the named pipe facilities are in InfoSphere DataStage Developer's Help.

  Select **Stage uses filter commands** if you want to specify a filter command to process data on the input or output links. Details of the actual command are specified on the Inputs page or Outputs page **General** tab (see "Defining Sequential File Input Data" on page 44 and "Defining Sequential File Output Data" on page 46).

  The **Stage uses named pipes** and **Stage uses filter commands** options are mutually exclusive.

If NLS is enabled, the **NLS** tab allows you to define character set mapping and Unicode settings for the stage. For more information, see "Defining Character Set Maps."

- **Inputs**. Contains information about the file formats and column definitions for each data input link. This page is displayed only if you have an input link to this stage.
- **Outputs**. Contains information about the file format and column definitions for the data output links. This page is displayed only if you have an output link from this stage.

Click **OK** to close this dialog box. Changes are saved when you save the job.

Defining Character Set Maps

You can define a character set map for a Sequential File stage using the **NLS** tab in the Sequential File Stage dialog box.

The default character set map (defined for the project or the job) can be changed by selecting a map name from the list. The tab also has the following fields:

- **Show all maps**. Choose this to display all the maps supplied with IBM InfoSphere DataStage in the list. Maps cannot be used unless they have been loaded using the Administrator client.
- **Loaded maps only**. Displays the maps that are loaded and ready for use.
- **Use Job Parameter...** . Allows you to specify a character set map as a parameter to the job containing the stage. If the parameter has not yet been defined, you are prompted to define it from the Job Properties dialog box.
- **Use UNICODE map**. If you select this, the character set map is overridden, and all data is read and written in Unicode format with two bytes per character.
  - If **Byte swapped** is selected, the data is read or written with the lower-order byte first. For example, 0X0041 (that is, "A") is written as bytes 0X41,0X00. Otherwise it is written as 0X00,0X41.
  - If **First character is Byte Order Mark** is selected, the stage reads or writes the sequence 0XFE,0XFF if byte swapped, or 0XFF,0XFE if not byte swapped.

Defining Sequential File Input Data

When you write data to a sequential file, the Sequential File stage has an input link. The properties of this link and the column definitions of the data are defined on the Inputs page in the Sequential File Stage dialog box.

The Inputs page has the following field and three tabs:

- **Input name**. The name of the input link. Choose the link you want to edit from the **Input name** list. This list displays all the input links to the Sequential File stage.
- **General**. Displayed by default. Contains the following parameters:
  - **File name**. The path name of the file the data is written to. You can enter a job parameter to represent the file created during run time. For details about how to define job parameters, see *IBM InfoSphere DataStage and QualityStage Designer Client Guide*. You can also browse for the file. The file name will default to the link name if you do not specify one here.
  - **Filter command**. Here you can specify a filter program that will process the data before it is written to the file. This can be used, for example, to specify a zip program to compress the data. You can type in or browse for the filter program, and specify any command line arguments it requires in the text box. This text box is enabled only if you have selected the **Stage uses filter commands** check box on the **Stage** page **General** tab (see "Using a Sequential File Stage" on page 43). Note that, if you specify a filter command, data browsing is not available so the **View Data** button is disabled.
  - **Description**. Contains an optional description of the input link.

    The **General** tab also contains options that determine how the data is written to the file. These are displayed under the **Update action** area:

- **Overwrite existing file**. This is the default option. If this option is selected, the existing file is truncated and new data records are written to the empty file.
- **Append to existing file**. If you select this option, the data records are appended to the end of the existing file.
- **Backup existing file**. If you select this check box, a backup copy of the existing file is taken. The new data records are written based on whether you chose to append to or overwrite the existing file.

  Note: The backup can be used to reset the file if a job is stopped or aborted at run time. See *IBM InfoSphere DataStage and QualityStage Designer Client Guide* for more details.

- **Format**. Contains parameters that determine the format of the data in the file. There are up to four check boxes:
  - **Fixed-width columns**. If you select this check box, the data is written to the file in fixed-width columns. The width of each column is specified by the SQL display size (set in the **Display** column in the Columns grid). This option is cleared by default.
  - **First line is column names**. Select this check box if the first row of data in the file contains column names. This option is cleared by default, that is, the first row in the file contains data.
  - **Omit last new-line**. Select this check box if you want to remove the last newline character in the file. This option is cleared by default, that is, the newline character is not removed.
  - **Flush after every row**. This only appears if you have selected **Stage uses named pipes** on the Stage page. Selecting this check box causes data to be passed between the reader and writer of the pipe one record at a time.

    There are up to seven fields on the **Format** tab:
  - **Delimiter**. Only active if you have not specified fixed-width columns. Contains the delimiter that separates the data fields in the file. By default this field contains a comma. You can enter a single printable character or a decimal or hexadecimal number to represent the ASCII code for the character you want to use. Valid ASCII codes are in the range 1 to 253. Decimal values 1 through 9 must be preceded with a zero. Hexadecimal values must be prefixed with &h. Enter 000 to suppress the delimiter.
  - **Quote character**. Only active if you have not specified fixed-width columns. Contains the character used to enclose strings. By default this field contains a double quotation mark. You can enter a single printable character or a decimal or hexadecimal number to represent the ASCII code for the character you want to use. Valid ASCII codes are in the range 1 to 253. Decimal values 1 through 9 must be preceded with a zero. Hexadecimal values must be prefixed with &h. Enter 000 to suppress the quote character.
  - **Spaces between columns**. This field is only active when you select the **Fixed-width columns** check box. Contains a number to represent the number of spaces used between columns.
  - **Default NULL string**. Contains the default characters that are written to the file when a column contains an SQL null (this can be overridden for individual column definition in the **Columns** tab).
  - **Default padding**. Contains the character used to pad missing columns. This is # by default, but can be set to another character here to apply to all columns, or can be overridden for individual column definitions in the **Columns** tab.

    The following fields appear only if you have selected **Stage uses named pipes** on the Stage page:
  - **Wait for reader timeout**. Specifies how long the stage will wait for a connection when reading from a pipe before timing out. Recommended values are from 30 to 600 seconds. If the stage times out, an error is raised and the job is aborted.
  - **Write timeout**. Specifies how long the stage will attempt to write data to a pipe before timing out. Recommended values are from 30 to 600 seconds. If the stage times out, an error is raised and the job is aborted.
- **Columns**. Contains the column definitions for the data on the chosen input link. In addition to the standard column definition fields (Column name, Key, SQL Type, Length, Scale, Nullable, Display, Data Element, and Description), Sequential File stage **Column** tabs also have the following fields:

– **Null string**. Fill this in if you want to override the default setting on the **Format** tab for this particular column.
– **Padding**. Fill this in if you want to override the default setting on the **Format** tab for this particular column.
– **Contains terminators**. Does not apply to input links.
– **Incomplete column**. Does not apply to input links.

  Note that the Scale for a sequential file column has a practical limit of 14. If values higher than this are used the results might be ambiguous.

  The SQL data type properties affect how data is written to a sequential file. The SQL display size determines the size of fixed-width columns. The SQL data type determines how the data is justified in a column: character data types are quoted and left justified, numeric data types are not quoted and are right justified. The SQL properties are in the Columns grid when you edit an input column.

Click **View Data...** to open the Data Browser. This enables you to look at the data associated with the input link. For a description of the Data Browser, see *IBM InfoSphere DataStage and QualityStage Designer Client Guide*.

Defining Sequential File Output Data

When you extract (read) data from a sequential file, the Sequential File stage has an output link. The properties of this link and the column definitions of the data are defined on the Outputs page in the Sequential File Stage dialog box.

The Outputs page has the following field and three tabs:

- **Output name**. The name of the output link. Choose the link you want to edit from the **Output name** list. This list displays all the output links to the Sequential File stage.
- **General**. Displayed by default. There are two fields:
  – **File name**. The path name of the file the data is extracted from. You can enter a job parameter to represent the file created during run time. You can also browse for the file.
  – **Filter command**. Here you can specify a filter program for processing the file you are extracting data from. This feature can be used, for example, to unzip a compressed file before reading it. You can type in or browse for the filter program, and specify any command line arguments it requires in the text box. This text box is enabled only if you have selected the **Stage uses filter commands** check box on the **Stage** page **General** tab (see "Using a Sequential File Stage" on page 43). Note that, if you specify a filter command, data browsing is not available so the **View Data** button is disabled.
  – **Description**. Contains an optional description of the output link.
- **Format**. Contains parameters that determine the format of the data in the file. There are three check boxes:
  – **Fixed-width columns**. If you select this check box, the data is extracted from the file in fixed-width columns. The width of each column is specified by the SQL display size (set in the **Display** column in the Columns grid). This option is cleared by default.
  – **First line is column names**. Select this check box if the first row of data in the file contains column names. This option is cleared by default, that is, the first row in the file contains data.
  – **Suppress row truncation warnings**. If the sequential file being read contains more columns that you have defined, you will normally receive warnings about overlong rows when the job is run. If you want to suppress these message (for example, you might only be interested in the first three columns and happy to ignore the rest), select this check box.

    There are up to eight fields on the **Format** tab:
  – **Missing columns action.** Allows you to specify the action to take when a column is missing from the input data. Choose **Pad with SQL null**, **Map empty string**, or **Pad with empty string** from the list.

- **Delimiter**. Only active if you have not specified fixed-width columns. Contains the delimiter that separates the data fields in the file. By default this field contains a comma. You can enter a single printable character or a decimal or hexadecimal number to represent the ASCII code for the character you want to use. Valid ASCII codes are in the range 1 to 253. Decimal values 1 through 9 must be preceded with a zero. Hexadecimal values must be prefixed with &h. Enter 000 to suppress the delimiter.
- **Quote character**. Only active if you have not specified fixed-width columns. Contains the character used to enclose strings. By default this field contains a double quotation mark. You can enter a single printable character or a decimal or hexadecimal number to represent the ASCII code for the character you want to use. Valid ASCII codes are in the range 1 to 253. Decimal values 1 through 9 must be preceded with a zero. Hexadecimal values must be prefixed with &h. Enter 000 to suppress the quote character.
- **Spaces between columns**. This field is only active when you select the **Fixed-width columns** check box. Contains a number to represent the number of spaces used between columns.
- **Default NULL string**. Contains characters which, when encountered in a sequential file being read, are interpreted as the SQL null value (this can be overridden for individual column definitions in the **Columns** tab).
- **Default padding**. Contains the character used to pad missing columns. This is # by default, but can be set to another character here to apply to all columns, or can be overridden for individual column definitions in the **Columns** tab.

  The following fields appear only if you have selected **Stage uses named pipes** on the Stage page:
- **Wait for writer timeout**. Specifies how long the stage will wait for a connection when writing to a pipe before timing out. Recommended values are from 30 to 600 seconds. If the stage times out, an error is raised and the job is aborted.
- **Read timeout**. Specifies how long the stage will attempt to read data from a pipe before timing out. Recommended values are from 30 to 600 seconds. If the stage times out, an error is raised and the job is aborted.

- **Columns**. Contains the column definitions for the data on the chosen output link. In addition to the standard column definition fields (Column name, Key, SQL Type, Length, Scale, Nullable, Display, Data Element, and Description), Sequential File stage **Column** tabs also have the following fields:
  - **Null string**. Fill this in if you want to override the default setting on the **Format** tab for this particular column.
  - **Padding**. Fill this in if you want to override the default setting on the **Format** tab for this particular column.
  - **Contains terminators**. Use this to specify how End of Record (EOR) marks are treated in this column. Choose from:

    Yes to specify that the data might include EOR marks and they should not be treated as meaning end of record. For the final column definition for a CSV file, the **Yes** option is disabled.

    Quoted to specify that any EOR marks that are part of the data are quoted, while unquoted EOR marks should be interpreted as end of record.

    No to specify that any EOR marks in the column should be interpreted as end of record.
  - **Incomplete column**. Allows you to specify the action taken if the column contains insufficient data to match the metadata. Choose from:

    Error to abort the job as soon as such a row is found.

    Discard & Warn to discard the current data row and issue a warning.

    Replace & Warn to pad a short column with SQL null, or act in accordance with **Missing columns action** if missing, and write a warning to the log file.

    Retain & Warn to pass the data on as it is, but issue a warning.

    Retain to pass the data on as it is.

    Replace to pad a short column with SQL null, or act in accordance with **Missing columns action** if missing.

The behavior of **Incomplete column** also depends on whether the sequential file is fixed-width or CSV. In CSV format it is impossible to have a short column, so the option applies only to missing columns and the **Retain** options have no meaning.

Click **View Data...** to open the Data Browser. This enables you to look at the data associated with the output link.

How the Sequential Stage Behaves

The following tables show how a Sequential File stage processes two rows of data with various options set in the stage editor.

The metadata for the link specifies that the data is organized in three columns containing three characters each. In the table, <EMPTY> indicates one of SQL null, empty string, or mapped empty string, depending on the settings.

Input Data Set 1

Row 1: ABC|123|<LF>YZ<LF>

Row 2: PQR...

Table 3. Input Data Set 1

Format Options	Column Options: (applies to all rows)		Output Data (first row)	Log Entries	Start of 2nd Row?	Comment
Line Termination	**Contains Terminators**	**Incomplete Column**				
<LF> UNIX	N	Error	None	Fatal Error	n/a	No output data
		Discard & Warn	None	Warning	Y	Row 1 discarded
		Replace & Warn	ABC|123|<EMPTY>	Warning	Y	Phantom row
		Replace	ABC|123|<EMPTY>	None	Y	Phantom row
		Retain & Warn	"ABC|123|" " " " "	Warning	Y	Phantom row
		Retain	"ABC|123|" " " " "	None	Y	Phantom row
<LF> UNIX	Y	Error	ABC|123|<LF>YZ	None	P	Correct data
		Discard & Warn	ABC|123|<LF>YZ	None	P	Correct data
		Replace & Warn	ABC|123|<LF>YZ	None	P	Correct data
		Replace	ABC|123|<LF>YZ	None	P	Correct data
		Retain & Warn	ABC|123|<LF>YZ	None	P	Correct data
		Retain	ABC|123|<LF>YZ	None	P	Correct data
<CR><LF> DOS	N	Error	ABC|123|<LF>YZ	None	n/a	No end-of-row
		Discard & Warn	ABC|123|<LF>YZ	None	n/a	No end-of-row
		Replace & Warn	ABC|123|<LF>YZ	None	n/a	No end-of-row
		Replace	ABC|123|<LF>YZ	None	n/a	No end-of-row
		Retain & Warn	ABC|123|<LF>YZ	None	n/a	No end-of-row
		Retain	ABC|123|<LF>YZ	None	n/a	No end-of-row
<CR><LF> DOS	Y	Error	ABC|123|<LF>YZ	None	n/a	No end-of-row
		Discard & Warn	ABC|123|<LF>YZ	None	n/a	No end-of-row
		Replace & Warn	ABC|123|<LF>YZ	None	n/a	No end-of-row
		Replace	ABC|123|<LF>YZ	None	n/a	No end-of-row
		Retain & Warn	ABC|123|<LF>YZ	None	n/a	No end-of-row
		Retain	ABC|123|<LF>YZ	None	n/a	No end-of-row
None	n/a	Error	ABC|123|<LF>YZ	None	<LF>	Data slip
		Discard & Warn	ABC|123|<LF>YZ	None	<LF>	Data slip
		Replace & Warn	ABC|123|<LF>YZ	None	<LF>	Data slip
		Replace	ABC|123|<LF>YZ	None	<LF>	Data slip
		Retain & Warn	ABC|123|<LF>YZ	None	<LF>	Data slip
		Retain	ABC|123|<LF>YZ	None	<LF>	Data slip

Input Data Set 2

Row 1: ABC|123|X<LF>Z<LF>

Row 2: PQR...

Table 4. Input Data Set 2

Format Options	Column Options: (applies to all rows)					
Line Termination	**Contains Terminators**	**Incomplete Column**	**Output Data (first row)**	**Log Entries**	**Start of 2nd Row?**	**Comment**
<LF> UNIX	N	Error	None	Fatal Error	n/a	No output data
		Discard & Warn	None	Warning	Z	Row 1 discarded
		Replace & Warn	ABC|123|<EMPTY>	Warning	Z	Phantom row
		Replace	ABC|123|<EMPTY>	None	Z	Phantom row
		Retain & Warn	ABC|123|X	Warning	Z	Phantom row
		Retain	ABC|123|X	None	Z	Phantom row
<LF> UNIX	Y	Error	ABC|123|X<LF>Z	None	P	Correct data
		Discard & Warn	ABC|123|X<LF>Z	None	P	Correct data
		Replace & Warn	ABC|123|X<LF>Z	None	P	Correct data
		Replace	ABC|123|X<LF>Z	None	P	Correct data
		Retain & Warn	ABC|123|X<LF>Z	None	P	Correct data
		Retain	ABC|123|X<LF>Z	None	P	Correct data
<CR><LF> DOS	N	Error	ABC|123|X<LF>Z	None	n/a	No end-of-row
		Discard & Warn	ABC|123|X<LF>Z	None	n/a	No end-of-row
		Replace & Warn	ABC|123|X<LF>Z	None	n/a	No end-of-row
		Replace	ABC|123|X<LF>Z	None	n/a	No end-of-row
		Retain & Warn	ABC|123|X<LF>Z	None	n/a	No end-of-row
		Retain	ABC|123|X<LF>Z	None	n/a	No end-of-row
<CR><LF> DOS	Y	Error	ABC|123|X<LF>Z	None	n/a	No end-of-row
		Discard & Warn	ABC|123|X<LF>Z	None	n/a	No end-of-row
		Replace & Warn	ABC|123|X<LF>Z	None	n/a	No end-of-row
		Replace	ABC|123|X<LF>Z	None	n/a	No end-of-row
		Retain & Warn	ABC|123|X<LF>Z	None	n/a	No end-of-row
		Retain	ABC|123|X<LF>Z	None	n/a	No end-of-row
None	n/a	Error	ABC|123|X<LF>Z	None	<LF>	Data slip
		Discard & Warn	ABC|123|X<LF>Z	None	<LF>	Data slip
		Replace & Warn	ABC|123|X<LF>Z	None	<LF>	Data slip
		Replace	ABC|123|X<LF>Z	None	<LF>	Data slip
		Retain & Warn	ABC|123|X<LF>Z	None	<LF>	Data slip
		Retain	ABC|123|X<LF>Z	None	<LF>	Data slip

Input Data Set 3

Row 1: ABC|12<LF>|XYZ<LF>

Row 2: PQR...

Table 5. Input Data Set 3

Format Options	Column Options: (applies to all rows)					
Line Termination	**Contains Terminators**	**Incomplete Column**	**Output Data (first row)**	**Log Entries**	**Start of 2nd Row?**	**Comment**
<LF> UNIX	N	Error	None	Fatal Error	n/a	No output data
		Discard & Warn	None	Warning	X	Discard row 1
		Replace & Warn	ABC|<EMPTY>|<EMPTY>	Warning	X	Phantom row
		Replace	ABC|<EMPTY>|<EMPTY>	None	X	Phantom row
		Retain & Warn	ABC|12|<EMPTY>	Warning	X	Phantom row
		Retain	ABC|12|<EMPTY>	None	X	Phantom row

Table 5. Input Data Set 3 (continued)

Format Options	Column Options: (applies to all rows)					
Line Termination	**Contains Terminators**	**Incomplete Column**	**Output Data (first row)**	**Log Entries**	**Start of 2nd Row?**	**Comment**
<LF> UNIX	Y	Error	ABC|12<LF>|XYZ	None	P	Correct data
		Discard & Warn	ABC|12<LF>|XYZ	None	P	Correct data
		Replace & Warn	ABC|12<LF>|XYZ	None	P	Correct data
		Replace	ABC|12<LF>|XYZ	None	P	Correct data
		Retain & Warn	ABC|12<LF>|XYZ	None	P	Correct data
		Retain	ABC|12<LF>|XYZ	None	P	Correct data
<CR><LF> DOS	N	Error	ABC|12<LF>|XYZ	None	n/a	No end-of-row
		Discard & Warn	ABC|12<LF>|XYZ	None	n/a	No end-of-row
		Replace & Warn	ABC|12<LF>|XYZ	None	n/a	No end-of-row
		Replace	ABC|12<LF>|XYZ	None	n/a	No end-of-row
		Retain & Warn	ABC|12<LF>|XYZ	None	n/a	No end-of-row
		Retain	ABC|12<LF>|XYZ	None	n/a	No end-of-row
<CR><LF> DOS	Y	Error	ABC|12<LF>|XYZ	None	n/a	No end-of-row
		Discard & Warn	ABC|12<LF>|XYZ	None	n/a	No end-of-row
		Replace & Warn	ABC|12<LF>|XYZ	None	n/a	No end-of-row
		Replace	ABC|12<LF>|XYZ	None	n/a	No end-of-row
		Retain & Warn	ABC|12<LF>|XYZ	None	n/a	No end-of-row
		Retain	ABC|12<LF>|XYZ	None	n/a	No end-of-row
None	n/a	Error	ABC|12<LF>|XYZ	None	<LF>	Data slip
		Discard & Warn	ABC|12<LF>|XYZ	None	<LF>	Data slip
		Replace & Warn	ABC|12<LF>|XYZ	None	<LF>	Data slip
		Replace	ABC|12<LF>|XYZ	None	<LF>	Data slip
		Retain & Warn	ABC|12<LF>|XYZ	None	<LF>	Data slip
		Retain	ABC|12<LF>|XYZ	None	<LF>	Data slip

Input Data Set 4

Row 1: ABC|123|<eof>

Table 6. Input Data Set 4

Format Options	Column Options: (applies to all rows)					
Line Termination	**Contains Terminators**	**Incomplete Column**	**Output Data (first row)**	**Log Entries**	**Start of 2nd Row?**	**Comment**
<LF> UNIX	N	Error	None	Fatal Error	n/a	No output data
		Discard & Warn	None	Warning		Discard row 1
		Replace & Warn	ABC|123|<EMPTY>	Warning		Correct data
		Replace	ABC|123|<EMPTY>	None		Correct data
		Retain & Warn	"ABC|123|" " " " "	Warning		Correct data
		Retain	"ABC|123|" " " " "	None		Correct data
<LF> UNIX	Y	Error	None	Fatal Error	n/a	No output data
		Discard & Warn	None	Warning		Discard row 1
		Replace & Warn	ABC|123|<EMPTY>	Warning		Correct data
		Replace	ABC|123|<EMPTY>	None		Correct data
		Retain & Warn	"ABC|123|" " " " "	Warning		Correct data
		Retain	"ABC|123|" " " " "	None		Correct data

Table 6. Input Data Set 4 (continued)

Format Options	Column Options: (applies to all rows)		Output Data (first row)	Log Entries	Start of 2nd Row?	Comment
Line Termination	**Contains Terminators**	**Incomplete Column**				
<CR><LF> DOS	N	Error Discard & Warn Replace & Warn Replace Retain & Warn Retain	None None ABC|123|<EMPTY> ABC|123|<EMPTY> "ABC|123|" " " " " "ABC|123|" " " " "	Fatal Error Warning Warning None Warning None	n/a	No output data Discard row 1 Correct data Correct data Correct data Correct data
<CR><LF> DOS	Y	Error Discard & Warn Replace & Warn Replace Retain & Warn Retain	None None ABC|123|<EMPTY> ABC|123|<EMPTY> "ABC|123|" " " " " "ABC|123|" " " " "	Fatal Error Warning Warning None Warning None	n/a	No output data Discard row 1 Correct data Correct data Correct data Correct data
None	n/a	Error Discard & Warn Replace & Warn Replace Retain & Warn Retain	None None ABC|123|<EMPTY> ABC|123|<EMPTY> "ABC|123|" " " " " "ABC|123|" " " " "	Fatal Error Warning Warning None Warning None	n/a	No output data Discard row 1 Correct data Correct data Correct data Correct data

Input Data Set 5

Row 1: ABC|12<CR>|<LF>YZ<CR><LF>

Row 2: PQR...

Table 7. Input Data Set 5

Format Options	Column Options: (applies to all rows)		Output Data (first row)	Log Entries	Start of 2nd Row?	Comment
Line Termination	**Contains Terminators**	**Incomplete Column**				
<LF> UNIX	N	Error Discard & Warn Replace & Warn Replace Retain & Warn Retain	None None ABC|12<CR>|<EMPTY> ABC|12<CR>|<EMPTY> "ABC|12<CR>|" " " " " "ABC|12<CR>|" " " " "	Fatal Error Warning Warning None Warning None	n/a Y Y Y Y Y	No output data Discard row 1 Phantom row Phantom row Phantom row Phantom row
<LF> UNIX	Y	Error Discard & Warn Replace & Warn Replace Retain & Warn Retain	ABC|12<CR>|<LF>YZ ABC|12<CR>|<LF>YZ ABC|12<CR>|<LF>YZ ABC|12<CR>|<LF>YZ ABC|12<CR>|<LF>YZ ABC|12<CR>|<LF>YZ	None None None None None None	P P P P P P	Correct data Correct data Correct data Correct data Correct data Correct data
<CR><LF> DOS	N	Error Discard & Warn Replace & Warn Replace Retain & Warn Retain	None None ABC|<EMPTY>|<EMPTY> ABC|<EMPTY>|<EMPTY> ABC|12|<EMPTY> ABC|12|<EMPTY>	Fatal Error Warning Warning None Warning None	n/a Y Y Y Y Y	No output data Discard row 1 Phantom row Phantom row Phantom row Phantom row

Table 7. Input Data Set 5 (continued)

Format Options	Column Options: (applies to all rows)					
Line Termination	**Contains Terminators**	**Incomplete Column**	**Output Data (first row)**	**Log Entries**	**Start of 2nd Row?**	**Comment**
<CR><LF> DOS	Y	Error	ABC|12<CR>|<LF>YZ	None	P	Correct data
		Discard & Warn	ABC|12<CR>|<LF>YZ	None	P	Correct data
		Replace & Warn	ABC|12<CR>|<LF>YZ	None	P	Correct data
		Replace	ABC|12<CR>|<LF>YZ	None	P	Correct data
		Retain & Warn	ABC|12<CR>|<LF>YZ	None	P	Correct data
		Retain	ABC|12<CR>|<LF>YZ	None	P	Correct data
None	n/a	Error	ABC|12<CR>|<LF>YZ	None	<CR>	Data slip
		Discard & Warn	ABC|12<CR>|<LF>YZ	None	<CR>	Data slip
		Replace & Warn	ABC|12<CR>|<LF>YZ	None	<CR>	Data slip
		Replace	ABC|12<CR>|<LF>YZ	None	<CR>	Data slip
		Retain & Warn	ABC|12<CR>|<LF>YZ	None	<CR>	Data slip
		Retain	ABC|12<CR>|<LF>YZ	None	<CR>	Data slip

Aggregator Stages

Aggregator stages classify data rows from a single input link into groups and compute totals or other aggregate functions for each group. The summed totals for each group are output from the stage via an output link.

Using an Aggregator Stage

If you want to aggregate the input data in a number of different ways, you can have several output links, each specifying a different set of properties to define how the input data is grouped and summarized.

When you edit an Aggregator stage, the Aggregator Stage dialog box appears. This dialog box has three pages:

- **Stage**. Displays the name of the stage you are editing. This page has a **General** tab which contains an optional description of the stage and names of before- and after-stage routines. For more details about these routines, see "Before-Stage and After-Stage Subroutines."
- **Inputs**. Specifies the column definitions for the data input link.
- **Outputs**. Specifies the column definitions for the data output link.

Click **OK** to close this dialog box. Changes are saved when you save the job.

Before-Stage and After-Stage Subroutines

The **General** tab on the Stage page contains optional fields that allow you to define routines to use, which are executed before or after the stage has processed the data.

- **Before-stage subroutine** and **Input Value**. Contain the name (and value) of a subroutine that is executed before the stage starts to process any data. For example, you can specify a routine that prepares the data before processing starts.
- **After-stage subroutine** and **Input Value**. Contain the name (and value) of a subroutine that is executed after the stage has processed the data. For example, you can specify a routine that sends an electronic message when the stage has finished.

Choose a routine from the list. This list contains all the routines defined as a **Before/After Subroutine** in the **Routines** folder in the repository tree. Enter an appropriate value for the routine's input argument in the **Input Value** field.

If you choose a routine that is defined in the repository, but which was edited but not compiled, a warning message reminds you to compile the routine when you close the **Aggregator Stage** dialog box.

A return code of 0 from the routine indicates success, any other code indicates failure and causes a fatal error when the job is run.

If you installed or imported a job, the **Before-stage subroutine** or **After-stage subroutine** field might reference a routine that does not exist on your system. In this case, a warning message appears when you close the Aggregator Stage dialog box. You must install or import the "missing" routine or choose an alternative one to use.

Defining Aggregator Input Data

Data to be aggregated is passed from a previous stage in the job design and into the Aggregator stage via a single input link. The properties of this link and the column definitions of the data are defined on the Inputs page in the Aggregator Stage dialog box.

Note: The Aggregator stage does not preserve the order of input rows, even when the incoming data is already sorted.

The Inputs page has the following field and two tabs:

- **Input name**. The name of the input link to the Aggregator stage.
- **General**. Displayed by default. Contains an optional description of the link.
- **Columns**. Contains a grid displaying the column definitions for the data being written to the stage, and an optional sort order.
  - **Column name**. The name of the column.
  - **Sort**. Displays the sort key position of the column, if sorting is enabled. For more information, see "Defining the Input Column Sort Order."
  - **Sort Order**. Specifies the sort order. This field is blank by default, that is, there is no sort order. Choose **Ascending** for ascending order, **Descending** for descending order, or **Ignore** if you do not want the order to be checked.
  - **Key**. Indicates whether the column is part of the primary key.
  - **SQL type**. The SQL data type.
  - **Length**. The data precision. This is the length for CHAR data and the maximum length for VARCHAR data.
  - **Scale**. The data scale factor.
  - **Nullable**. Specifies whether the column can contain null values.
  - **Display**. The maximum number of characters required to display the column data.
  - **Data element**. The type of data in the column.
  - **Description**. A text description of the column.

Defining the Input Column Sort Order

When the Aggregator stage collates input data for aggregating, it is stored in memory. If one or more group columns in the input data are sorted, this can greatly improve the way in which the Aggregator stage handles the data.

Sorted input data can be output from an ODBC or a UniVerse stage (using an ORDER BY clause in the SQL statement) or a Sequential File stage.

To use sorted input data, you can use the additional column properties on the **Columns** tab on the Inputs page.

Enter a number in the **Sort** column specifying the position that column has in the sort key. For example, if the input data was sorted on a date then on a product code, the sort key position for the date column would 1 and the sort key position for the product code column would be 2. A value of 1 always indicates the most significant key. If you do not specify a value in this field, the column is added to the end of the sort key sequence. When you click **OK**, all the columns are sorted in sequence from the most significant column upward.

Choose the order in which the data is sorted from the **Sort Order** column. The default setting is none:

- **Ascending**. Choose this option if the input data in the specified column is sorted in ascending order. If you choose this option, the IBM InfoSphere DataStage server checks the order at run time.
- **Descending**. Choose this option if the input data in the specified column is sorted in descending order. If you choose this option, the InfoSphere DataStage server checks the order at run time.
- **Ignore**. Do not check order. Choose this option if the sort order used by the input data is not simply ascending or descending order, but uses a more complex sort order. You must take care when choosing this option. At run time the InfoSphere DataStage server does not check the sort order of the data, which might cause erroneous errors. If you choose this option, a warning message appears when you click **OK**. You must acknowledge this message before you can edit other input columns.

Defining Aggregator Output Data

When you output data from an Aggregator stage, the properties of output links and the column definitions of the data are defined on the Outputs page in the Aggregator Stage dialog box.

The Outputs page has the following field and two tabs:

- **Output name**. The name of the output link. Choose the link to edit from the **Output name** list. This list displays all the output links from the stage.
- **General**. Displayed by default. Contains an optional description of the link.
- **Columns**. Contains a grid displaying the column definitions for the data being output from the stage. The grid has the following columns:
  - **Column name**. The name of the column.
  - **Group**. Specifies whether to group by the data in the column.
  - **Derivation**. Contains an expression specifying how the data is aggregated. This is a complex cell, requiring more than one piece of information. Double-clicking the cell opens the Derivation dialog box. For more information, see "Aggregating Data."
  - **Key**. Indicates whether the column is part of the primary key.
  - **SQL type**. The SQL data type.
  - **Length**. The data precision. This is the length for CHAR data and the maximum length for VARCHAR data.
  - **Scale**. The data scale factor.
  - **Nullable**. Specifies whether the column can contain null values.
  - **Display**. The maximum number of characters required to display the column data.
  - **Data element**. The type of data in the column.
  - **Description**. A text description of the column.

    For a description of how to enter and edit column definitions, see *IBM InfoSphere DataStage and QualityStage Designer Client Guide*.

Aggregating Data

The data sources you are extracting data from can contain many thousands of rows of data. For example, the data in a sales database can contain information about each transaction or sale. You could pass all this

data into your data warehouse. However, this would mean you would have to search through large volumes of data in the data warehouse before you get the results you need.

If you only want summary information, for example, the total of product A sold since 01/01/96, you can aggregate your data and only pass the summed total to the data warehouse. This reduces the amount of data you store in the data warehouse, speeds up the time taken to find the data you want, and ensures the data warehouse stores data in a format you need.

The Aggregator stage allows you to group by or summarize any columns on any of the output links.

Note: Every column output from an Aggregator stage must be either grouped by or summarized.

A group of input data is a set of input rows that share the same values for all the grouped by columns. For example, if your sales database contained information about three different products, A, B, and C, you could group by the **Product** column. All the information about product A would be grouped together, as would all the information for products B and C.

By summarizing data, you can perform basic calculations on the values in a particular column. The actions you can perform depend on the SQL data type of the selected column.

For numeric SQL data types you can perform the following actions:

- Minimum. Returns the lowest value in the column.
- Maximum. Returns the highest value in the column.
- Count. Counts the number of values in the column.
- Sum. Totals the values in the column.
- Average. Averages the values in the column.
- First. Returns the first value in the column.
- Last. Returns the last value in the column.
- Standard Deviation. Returns the standard deviation of the values in the column.

  In calculating Standard Deviation, IBM InfoSphere DataStage uses the formula:

  $standarddeviation = \mathrm{sqrt}\ [\ (\mathrm{sum}(Xi^2) - N\ \mathrm{avg}(Xi)^2) / N\ ]$

  Some other packages, such as Microsoft Excel, use the formula:

  $standarddeviation = \mathrm{sqrt}\ [\ (\mathrm{sum}(Xi^2) - N\ \mathrm{avg}(Xi)^2) / (N-1)]$

For any other SQL data types you can perform the following actions:

- Minimum. Returns the lowest value in the column.
- Maximum. Returns the highest value in the column.
- Count. Counts the number of values in the column.
- First. Returns the first value in the column.
- Last. Returns the last value in the column.

For example, if you want to know the total number of product A sold, you would sum the values in the **QtySold** column.

To group by or summarize a column, you must edit the **Derivation** column in the Output Column dialog box. Do this by double-clicking the cell to open the Derivation dialog box.

The Derivation dialog box contains the following fields and option:

- **Source column**. Contains the name of the column you want to group by or summarize, in the format *linkname.columnname*. You can choose any of the input columns from the list.

- **Aggregate function**. Contains the aggregation function to perform. Choose the function you want from the list. The default option is **Sum**.
- **Group by this column**. Specifies whether the column will be grouped. This check box is cleared by default.

If you want to group by the column, select the **Group by this column** check box. The aggregate function is automatically set to **(grouped)**, and you cannot select an aggregate function from the list.

To use an aggregate function, clear the **Group by this column** check box and select the function you want to use from the **Aggregate function** list.

Click **OK** to save the settings for the column.

Command Stage

Command Stage is an active stage that can execute various external commands, including server engine commands, programs, and jobs from anywhere in the IBM InfoSphere DataStage data flow. You can execute any command, including its arguments, that you can type to the shell of the operating system, such as Windows or UNIX. Examples include Perl scripts, DOS batch files, UNIX scripts, and other command-line executable programs that you can call if they are not interactive.

A graphical user interface (GUI) is available for Command Stage.

You can use Command Stage anywhere in a job path to invoke an external command. The before- and after-routines that are already available act similarly, except that you can put Command Stage anywhere in a job stream and call it multiple times in parallel.

If the stage is placed midstream and **Do not forward row data** is not selected, it moves the data to the output link. If the stage is at the end of a path, it executes the command and passes the incoming data through unaltered. The arrival of the row merely causes the command execution.

Command Stage can have only one input and one output link:
- **Input link.** Specifies a row of actual data or a single row from a previous instance of Command Stage. You can place a Command Stage midstream or at the end of a job path (with no output link).
- **Output link.** If you run Command Stage at the beginning of a job path for an output link, the stage executes the specified command and sends a single row down the output link. Minimally, this row contains the return code from the specified command in the first column. A Transformer stage can then use InfoSphere DataStage branching operations to process this code. If **Output to link** is selected, the second column holds the output for the command.

The GUI handles the creation of columns on the output link by examining the values of **Output to link**, **Do not forward row data**, and **Do not wait for command**.

Functionality

Supported Functionality

Command Stage has the following functionality:
- More flexibility than using other before- or after-stage routines.
- Visual and textual metadata.
- Graphical invocation of external commands without resorting to job control coding.
- Easier processing of return codes from external commands.
- The stage and its links appear as event metadata within the IBM InfoSphere DataStage Suite Metadata Management Services.

- Support for NLS (National Language Support).

Unsupported Functionality

The following functionality is not supported:

- Data transformation capabilities on rows flowing through the stage. Use the Transformer and the Aggregator stages to do this.
- Commands requiring user-input or creation of windows. They cause job failures.
- Client access to an RDBMS. If you want to execute an SQL statement, use calls to existing client applications, including InfoSphere DataStage jobs.
- Direct access to server engine commands. You cannot use this stage to return rows that are generated as a result of command execution to the engine.

Terminology

The following list describes the Command Stage terms used in this document:

Term Description

Before and after routines
The external routines that you can define to be called before a job begins and after a job exits. Write these routines in IBM InfoSphere DataStage BASIC.

Some stages support before- and after-stage routines. These are called before or after a stage is invoked.

ExecTCL
A built-in routine that executes server engine commands from an InfoSphere DataStage job.

ExecDOS
A built-in routine that executes DOS commands from an InfoSphere DataStage job.

Using Command Stage

When you use the GUI to edit a Command Stage, the Command Stage dialog box opens. This dialog box has the **Stage**, **Input**, and Output pages, depending on whether there are inputs to and outputs from the stage:

- **Stage.** This page displays the name of the stage you are editing. The **General** tab defines the command type, the text of the command, the action to take if errors occur, where to write the output, and whether the job waits for the command to complete. You can also describe the purpose of the stage. For details, see "Defining the Command."

  The **NLS** tab defines a character set map to use with the stage. This tab appears only if you have installed NLS for IBM InfoSphere DataStage. For details, see "Defining Character Set Mapping" on page 58.
- **Input.** This page is displayed only if you have an input link to this stage. It specifies when to execute the command and how to handle the rows from this link.
- **Output.** This page is displayed only if you have an output link to this stage. It specifies how to handle the output from the command.

Defining the Command

The command parameters are set on the **General** tab of the Stage page. Specify the appropriate information using the following fields:

- **Command type.** The type of command to be executed. Select one of the following options:
  - OS. The stage executes an operating system command.

  - TCL. The stage executes an server engine command. You can run IBM InfoSphere DataStage BASIC programs.

    For information about using these commands, see "Using Commands" on page 60.
- **Command.** The string to be passed as the command.
- **Abort if command fails.** If selected, the job aborts if errors occur while executing the command.
- **Disable output to log.** If selected, the output from the command is not written to the InfoSphere DataStage log.
- **Do not wait for command.** If selected, the job does not wait for the command to complete before continuing. The job is an independent process and continues to process the data. It executes the command as a thread on Windows. The stage waits if the command is still executing after all data is processed.

  Selecting this option removes the COMMAND.RTNCODE and COMMAND.OUTPUT data elements from the output link. Link output is disallowed, but the output and the return code for the command are still written to the InfoSphere DataStage log and the output file. The first column on the output link is not used for the return code.

  Additionally, the following options are disabled:
  - **Abort if command fails** (stage)
  - **Repeat for each row** (input)
  - **Execute command after row** (input)
  - **Do not forward row data** (input)
  - **Output to link** (output)
- **Output to file.** Writes output from the command to a file. If you do not specify a path name, the file is created in the home directory for the project. If you leave the field blank, no output file is created.
- **Description.** Optional. Describes the purpose of Command Stage.

Defining Character Set Mapping

You can define a character set map for a stage. Do this from the **NLS** tab on the Stage page. The **NLS** tab appears only if you have installed NLS.

Specify information using the following button and fields:
- **Map name to use with stage.** The default character set map is defined for the project or the job. You can change the map by selecting a map name from the list.
- **Use Job Parameter....** Specifies parameter values for the job. Use the format #*Param*#, where *Param* is the name of the job parameter. The string #*Param*# is replaced by the job parameter when the job is run.
- **Show all maps.** Lists all the maps that are shipped with IBM InfoSphere DataStage.
- **Loaded maps only.** Lists only the maps that are currently loaded.

Defining Command Stage Input Data

When a row of actual data or a single row from a previous instance of Command Stage arrives on an input link of this stage, it executes the specified command. Define the properties of this link and the column definitions of the data on the Input page in the Command Stage dialog box of the GUI.

About the Input Page

The Input page has an **Input name** field, the **General** and **Columns** tabs, and the **Columns...** button:
- **Input name.** The name of the input link. Choose the link you want to edit from the **Input name** drop-down list box. This list box displays all the input links to Command Stage.

- Click the **Columns...** button to display a brief list of the columns designated on the input link. As you enter detailed metadata in the **Columns** tab, you can leave this list displayed.

General Tab:

This tab, displayed by default, contains the following fields:

- **Repeat for each row.** If selected, executes the specified command for each row that arrives on this link. If **Do not wait for command** is selected from the General tab of the Stage page, this option is disabled to avoid overwhelming the server with processes.
- **Execute command after row.** If selected, executes the specified command after the row is copied and sent to the output link. If there is no output link, this option is disabled. By default, the command is executed asynchronously when the row arrives on the input link.

  Selecting this option removes the COMMAND.RTNCODE and COMMAND.OUTPUT data elements from the output link. Link output is disallowed, but the output and the return code for the command are still written to the IBM InfoSphere DataStage log and the output file. The following options are disabled:

  – **Do not forward row data** (input)
  – **Output to link** (output)
- **Do not forward row data.** If cleared, the stage passes rows through to the same number of columns on the output link, provided it contains both input and output links. You cannot select this option if no output link exists.

  If cleared, the column definitions are copied from the input link to the output link. The "Command stage pass thru column" label in the **Description** field identifies each copied column for removal.
- **Description.** Optional. Describes the purpose of the input link.

Columns Tab:

This tab contains the column definitions for the data written to the data source. The **Columns** tab behaves the same way as the **Columns** tab in the ODBC stage.

Defining Command Stage Output Data

You can write the output of a command as a column on an output link of Command Stage. The GUI automatically manages the output column definitions. The output columns depend more on your choices for field values than on the metadata requirements of their targets. Therefore, you have minimal flexibility in defining Command Stage output columns.

Passthrough columns must have the same data types and sizes as the corresponding input columns. However, you can edit the name, data element, derivation, and description fields for the columns.

About the Output Page

The **Output** page has an **Output name** field, the **General** and **Columns** tabs, and the **Columns...** button.

- **Output name.** The name of the output link. Choose the link you want to edit from the **Output name** drop-down list box. This list box displays all the output links.
- Click the **Columns...** button to display a brief list of the columns designated on this link. As you enter detailed metadata in the **Columns** tab, you can leave this list displayed.

General Tab:

This tab, displayed by default, contains the following fields:

- **Output to link.** If selected, sends the output from the command as the second column on the output link. This COMMAND.OUTPUT column holds the output of the command execution.

- **Description.** Optional. Describes the purpose of the output link.

Columns Tab:

This tab contains the column definitions for the data being output on the chosen link. The GUI automatically manages the output column definitions.

If **Do not wait for command** is not selected and **Output to link** output option is selected, the COMMAND.RTNCODE and COMMAND.OUTPUT data elements for the column definitions contain the return code and the command output respectively. However, the **Derivation** field is meaningless for this stage.

Using Commands

You can execute any command, including its arguments, that you can type to the shell of the operating system, for example, Perl scripts, DOS batch files, UNIX scripts, and other command-line driven programs that are not interactive or do not request input.

TCL Commands and BASIC Programs

You can set the Command type to TCL on the **General** tab of the Stage page to execute TCL commands and run BASIC programs.

dsjob Command

You can use the *dsjob* command to call other IBM InfoSphere DataStage jobs from Command Stage. InfoSphere DataStage provides the *dsjob* program to let you run compiled jobs from a command line instead of from InfoSphere DataStage. *dsjob* has the following simple syntax:

```
dsjob -run [ -mode ] [ -param ] [ -warn ] [ -rows ] [ -wait ]
[ -stop ] [ -jobstatus ] [ -userstatus ] project job
```

For full syntax information, see *InfoSphere DataStage Programmer's Guide*.

Note: If you select **Omit** in the Attach to Project dialog box as you start InfoSphere DataStage, you must use the -user and -password options when you use the *dsjob* command.

You can write the output of a command to any of the following:

- The InfoSphere DataStage log. This is the default.
- Output links. If output links exist, you can write the output as a column on the link, in addition to the return code for the command. The return code is automatically sent as the first column on the output link.
- A file.

  Note: Since the stage sends the return code for the command as the first column of an output link, the GUI provides for this automatically. If you use the standard grid editor, you must manually add the mandatory columns to the column definitions for the output link.

InterProcess Stages

An InterProcess (IPC) stage is a passive stage which provides a communication channel between IBM InfoSphere DataStage processes running simultaneously in the same job. It allows you to design jobs that run on SMP systems with great performance benefits. To understand the benefits of using IPC stages, you need to know a bit about how InfoSphere DataStage jobs actually run as processes. See "IBM InfoSphere DataStage Jobs and Processes" on page 5 for information.

The output link connecting IPC stage to the stage reading data can be opened as soon as the input link connected to the stage writing data has been opened.

You can use InterProcess stages to join passive stages together. For example you could use them to speed up data transfer between two data sources:

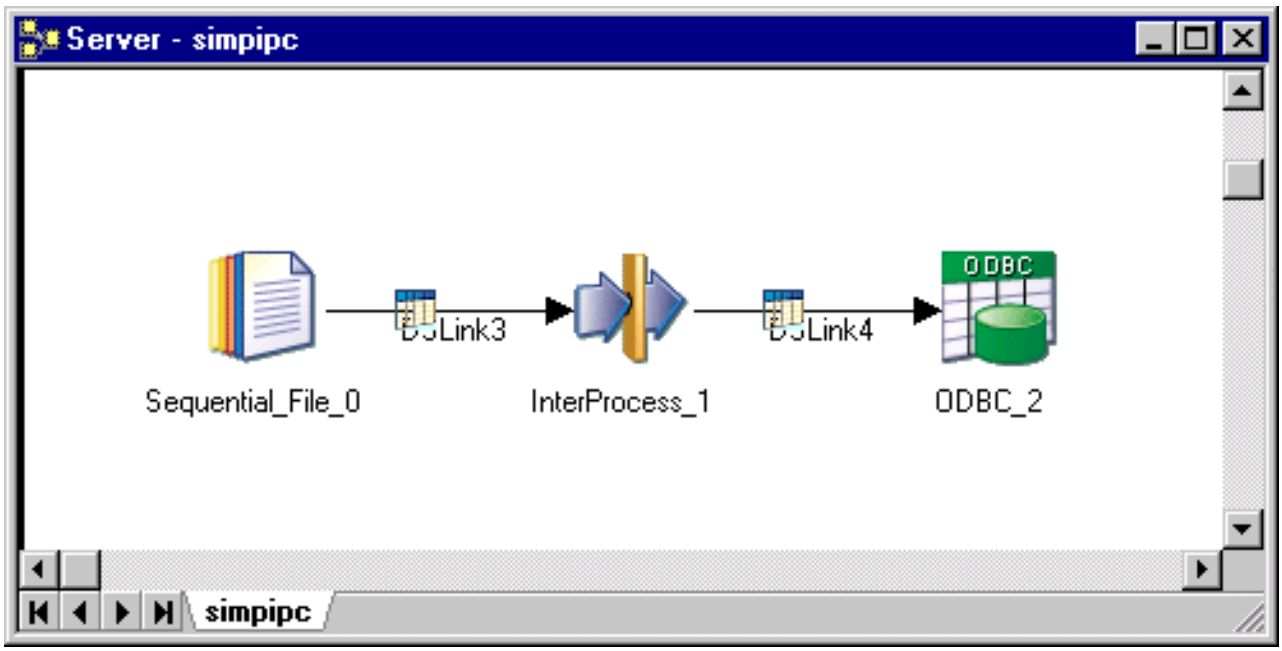


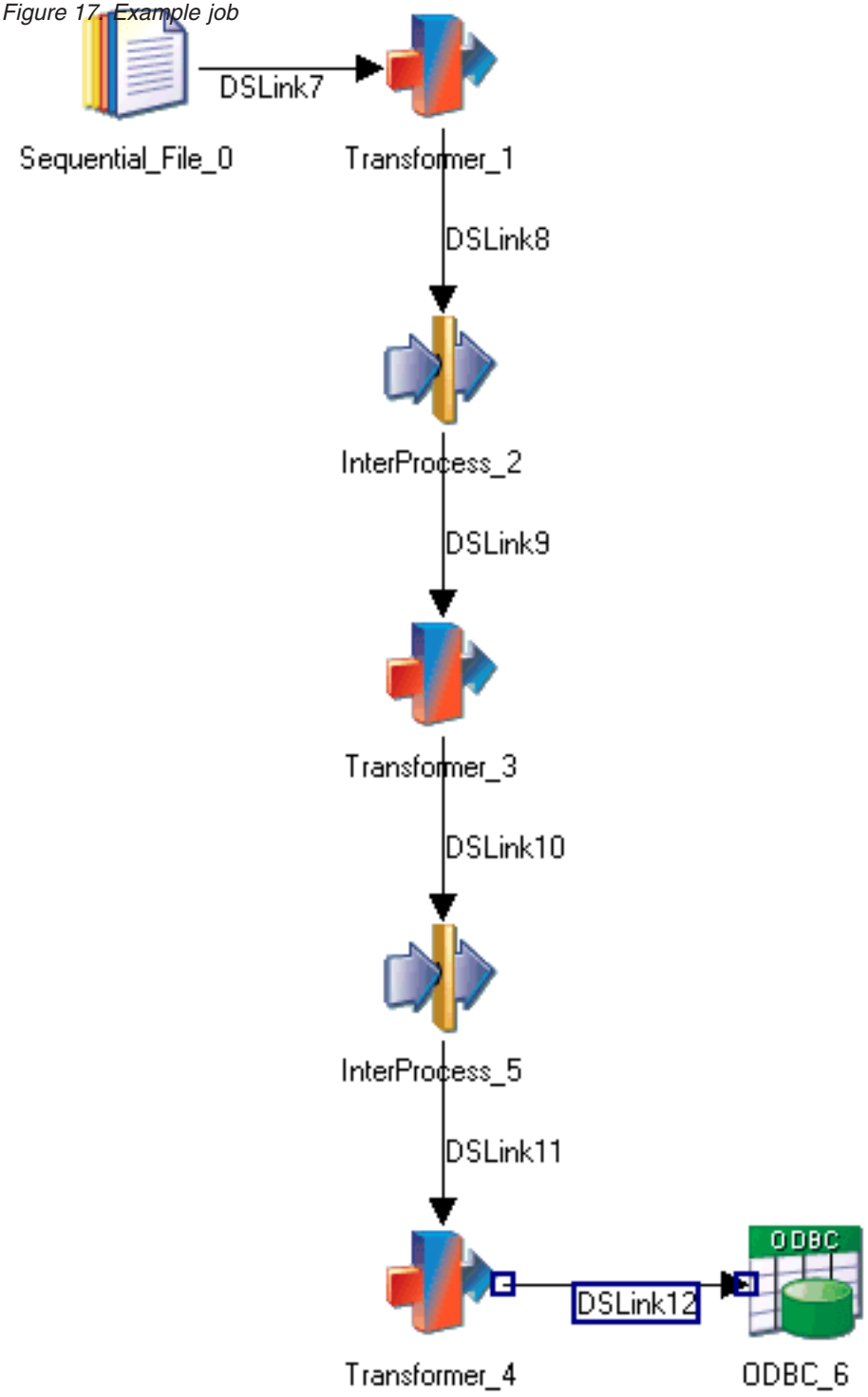
Figure 16. Example job

In this example the job will run as two processes, one handling the communication from the Sequential File stage to the IPC stage, and one handling communication from the IPC stage to the ODBC stage. As soon as the Sequential File stage has opened its output link, the IPC stage can start passing data to the ODBC stage. If the job is running on a multiprocessor system, the two processors can run simultaneously so the transfer will be much faster.

You can also use the IPC stage to explicitly specify that connected active stages should run as separate processes. This is advantageous for performance on multiprocessor systems. You can also specify this behavior implicitly by turning interprocess row buffering on, either for the whole project via the

Administrator client, or individually for a job in its Job Properties dialog box.

Figure 17. Example job



Using the IPC Stage

When you edit an IPC stage, the InterProcess Stage dialog box appears. This dialog box has three pages:

- **Stage**. The Stage page has two tabs, **General** and **Properties**. The General page allows you to specify an optional description of the stage. The **Properties** tab allows you to specify stage properties.
- **Inputs**. The IPC stage can only have one input link. The Inputs page displays information about that link.
- **Outputs**. The IPC stage can only have one output link. The Outputs page displays information about that link.

Defining IPC Stage Properties

The **Properties** tab allows you to specify two properties for the IPC stage:

- **Buffer Size**. Defaults to 128 Kb. The IPC stage uses two blocks of memory; one block can be written to while the other is read from. This property defines the size of each block, so that by default 256 Kb is allocated in total.
- **Timeout**. Defaults to 10 seconds. This gives a time limit for how long the stage will wait for a process to connect to it before timing out. This normally will not need changing, but might be important where you are prototyping multiprocessor jobs on single processor platforms and there are likely to be delays.

Defining IPC Stage Input Data

The IPC stage can have one input link. This is where the process that is writing connects.

The Inputs page has two tabs:

- **General**. The **General** tab allows you to specify an optional description of the stage.
- **Columns**. The **Columns** tab contains the column definitions for the data on the input link. This is normally populated by the metadata of the stage connecting on the input side. You can also **Load** a column definition from the repository, or type one in yourself (and **Save** it to the repository if required). Note that the metadata on the input link must be identical to the metadata on the output link.

Defining IPC Stage Output Data

The IPC stage can have one output link. This is where the process that is reading connects.

The Outputs page has two tabs: **General** and **Columns**.

- **General**. The **General** tab allows you to specify an optional description of the stage.
- **Columns**. The **Columns** tab contains the column definitions for the data on the output link. This is normally populated by the metadata of the stage connecting on the input side. You can also **Load** a column definition from the repository, or type one in yourself (and **Save** it to the repository if required). Note that the metadata on the output link must be identical to the metadata on the input link.

FTP Plug-in Stages

Like the Sequential File stage, the FTP Plug-in stage extracts data from, or writes data to, a single text file. However, the text files to be accessed by the FTP Plug-in stage reside on another machine (possibly with a different file system and character file storage conventions) over a communications network instead of on a local disk.

The FTP Plug-in stage provides users with rapid and efficient remote file access using existing FTP servers on remote platforms. The FTP Plug-in stage does not require additional installation on the remote platforms.

Additionally, the FTP Plug-in stage provides the option to execute before- and after-commands on the remote machine. This automates the following data flow processes:

- **Before it begins the file transfer.** You can use the before-command to prepare a file to be transferred or to prepare the remote machine to receive it.
- **After it completes the file transfer.** You can use the after-command to delete temporary files or to start a subsequent activity that uses the transferred file.

Each FTP Plug-in stage is a passive stage that can have any number of input and output links:

- Input links specify the data you are writing, which is a stream of rows to be loaded into a single remote file.
- Output links specify the data you are extracting, which is a stream of rows to be read from a single remote file.

Functionality

The FTP Plug-in stage has the following functionality and benefits:

- Shares common properties of the remote host name, user name, password, and directory path to or from which files are transferred for each stage instance.
- Corresponds generally to an independent file transfer session for each link, so that multiple files can be transferred concurrently.
- Acts as an FTP client, using a generic file transfer protocol to initiate sessions with and transfer files to or from any file transfer server. Retains an FTP session long enough to allow the transfer of large amounts of data.
- Supports the STREAM data protocol. If a STREAM transfer connection is closed, the job aborts with an error message.
- Handles job failures appropriately when incomplete files are transferred.

  Note: You can specify the number of rows to be processed by a job on the Limits page in the IBM InfoSphere DataStage Director client. As of Release 1.3, if you perform row limiting, fatal errors might be recorded in the log file at the end of the job because of premature closing of the data connection. However, the data transfer is completed for the number of rows selected.
- Supports a user-specified number of connection retries and retry intervals.
- Provides optional before- and after-commands to be run on the remote machine before and after a file is successfully transferred (requires a telnet server to use all capabilities on Windows).
- Provides an optional tracing level to diagnose performance issues.
- Lets you read or write ASCII or binary data.

  Note: Binary mode is not supported on the Parallel Server canvas. See the input property Data Representation Type and the output property Data Representation Type.
- Uses the stage and link properties and column type to determine the format for character strings before the transfer.
- Lets you control which process initiates the connection request for data transfer.
- Provides optional use of metadata definitions for reading a remote file.
- Lets you validate the existence of the remote file within the InfoSphere DataStage Director client (output link only).
- Supports NLS (National Language Support).

The following functionality is not supported:

- Bulk loading for stream input links
- Keyed lookups on a file transfer stage
- Stored procedures

Terminology

The following list explains FTP Plug-in terms used in this document:

Term Description

after-command
The command to be executed on the remote machine using a telnet session after the transfer is complete.

before-command
The command to be executed on the remote machine using a telnet session before starting the transfer.

FTP File Transfer Protocol. An interactive file transfer capability often used on TCP/IP networks.

rollback
Cancels all file I/O changes made during a transaction.

telnet The name of a protocol session that acts as a standard remote terminal emulation with communications to the host over a network.

transaction
A sequence of file I/O operations treated as one logical operation with respect to recovery and visibility to other users.

Installing the Stage

To specify transaction rollbacks, commits, or after/before processing to the Windows server, you must first provide a telnet server other than UniVerse telnet.

Properties

The tables in the following sections include the following column heads:

- **Prompt** is the text that the job designer sees in the stage editor user interface.
- **Type** is the data type of the property.
- **Default** is the text used if the job designer does not supply any value.
- **Description** describes the properties.

Stage Properties

The FTP Plug-in stage supports the following stage properties:

Prompt	Type	Default	Description
Server Name	String	None	Required. The name of the host machine for the FTP server on which the file resides.
Remote FTP Port	Long	21	Required. The port number of the remote machine's FTP server.
Remote Telnet Port	Long	23	Required. The port number of the remote machine's telnet server.
User Name	String	None	Required. The user name to log on to the remote machine.

Prompt	Type	Default	Description
User Password	String	None	The password for the specified user. Required if the remote machine uses a password for "User Name."
Account Name	String	None	The account name for the remote FTP login. Required only if the remote machine needs user account information during the login process.
Tracing Level	Long	0	Optional. Controls the type of tracing information that is added to the log. Use one of the following tracing levels: 0 No tracing 1 Report stage properties
Retries	Long	3	Optional. The number of retries if the connection fails.
Retry Interval	Long	15	Optional. The number of seconds to wait between retries if the connection fails.
Number of Telnet Prompts	String	2	Required if telnet services are being used. The total number of expected prompts that are received during the process of logging on to the telnet server.
Telnet Prompt 1	String	login	Required if telnet services are being used. The literal string (case-insensitive) that is sent by the telnet server, prompting the IBM InfoSphere DataStage process for login data.
Telnet Reply 1	String	None	Required if telnet services are being used. The telnet user name to log on to the telnet session.
Telnet Prompt 2	String	password	Required if telnet services are being used. The literal string (case-insensitive) that is sent by the telnet server, prompting the InfoSphere DataStage process for password data.
Telnet Reply 2	String	None	Required if telnet services are being used. The telnet password for the specified telnet user.

Prompt	Type	Default	Description
Telnet Prompt *n*	String	None	Any prompts that are needed to connect to a target system through telnet, in addition to login and password.
Telnet Reply *n*	String	None	Any replies that are needed to connect to a target system through telnet, in addition to login and password.
Command Timeout	Int	50	The number of milliseconds to wait for the Telnet Before and After Commands to complete.

Input Link Properties

The following table lists the input link properties in the grid editor:

Table 8. Input link properties

Prompt	Type	Default	Description
Remote Path	String	None	Optional. The path name of the working directory on the remote machine where the files to be retrieved or sent reside.
Remote File Name	String	None	Required. The name of the file on the remote machine to be retrieved or sent.
Data Representation Type	List	ASCII	Required. Controls how the data in the remote file is read or written. For **ASCII** representation, the data transfer uses standard NVT-ASCII, primarily for text files. For **binary** representation, the data is transferred in contiguous bits as IMAGE data. You must set "Fixed-width Columns" to Yes. **Note:** Binary mode is not supported when the stage is run on the Parallel canvas. To transfer data in binary mode, use data types of binary or varbinary with "Data Representation Type" set to ASCII.

Table 8. Input link properties (continued)

Prompt	Type	Default	Description
Line Termination	List	[CR] [LF] (DOS-Style Termination)	Specifies the row (end-of-line) termination sequence in the remote file. If "Data Representation Type" is set to ASCII, the valid values are no termination and [CR] [LF].
Fixed-width Columns	String	No	Required. Indicates whether the data in the remote file can be with fixed-width columns.
Spaces Between Columns	Long	0	The number of spaces between fixed-width columns in the remote file. Required if "Fixed-width Columns" is set to Yes.
Column Delimiter	Char	, (comma)	Required if "Fixed-width Columns" is set to No. The delimiter that separates the data fields in the remote file. You can enter single character without quotes or the ASCII value of the character you want to use.
Quote Character	Char	" (double quote)	Optional and only valid if "Fixed-width Columns" is set to No. The single character used to enclose a data value that contains the delimiter character as data. You can also enter the ASCII value for the character you want to use. You can suppress "Quote Character" by entering no value.
Escape Character	Char	\ (back- slash)	Required. The single character entered to be interpreted as the escape character.
Null String	String	None	Optional. Specifies the string that is to be interpreted as the SQL null value.
First Line Column Names	String	No	Required if "Data Representation Type" is set to ASCII. Specifies whether to transfer the first line in the remote file (that is, it might contain column names).

Table 8. Input link properties (continued)

Prompt	Type	Default	Description
Omit Last New Line	String	No	Required. Indicates whether you want to omit the last newline at the end of the data while sending it to the remote machine.
Append to File	String	No	Optional. Indicates whether the data is put into the remote file in append or overwrite mode. Yes indicates append to the existing file. No indicates overwrite the file.
Back Up File	String	No	Optional. Indicates whether "Telnet Backup Command" is executed before proceeding with the job.
Telnet Backup Command	String	None	Optional. Specifies the telnet command to execute on the remote machine before the job writes to the remote file. This telnet command is executed only if "Back Up File" is set to Yes. Use this command to create file backups.
Telnet Before Command	String	None	Optional. The telnet command to execute on the remote machine before starting a job.
Telnet After Command	String	None	Optional. Specifies the telnet command to execute on the remote machine after completing a job.
Transaction Begin Command	String	None	Optional. Specifies the telnet command to execute before starting the file transfer to the remote machine. Use this command to make temporary copies of files.
Transaction Commit Command	String	None	Optional. Specifies the telnet command to execute after a successful file transfer. Use this command to delete any temporary files created.

Table 8. Input link properties (continued)

Prompt	Type	Default	Description
Transaction Rollback Command	String	None	Optional. Specifies the telnet command to execute if an error occurs while sending the file to the remote machine, or if you use the Director client to reset the job. Use this command to restore any file from the temporary copy in the event of a failure or abort.
FTP Data Connection Mode	List	Passive	Specifies which process initiates the connection for the data transfer. If set to Active, connections are initiated by the FTP server. If set to Passive, connections are initiated from the host system where engine tier is installed. This lets you store files on remote hosts that are outside router-based firewall. **Digital OpenVMS systems.** Set to Active for input links so that the FTP server initiates the connection for data transfer. Otherwise, no data is accepted.

Link Tracing Level	Long	0	Optional. Controls the type of tracing information that is added to the log. The available tracing levels are:
			`0 No tracing`
			`1 Link properties`
			`2 Performance`
			`4 FTP messages`
			`8 Telnet messages`
			`16 Function tracing`
			`32 Telnet data dump`
			
			You can combine the tracing levels. For example, a tracing level of 3 means that link properties and performance messages are added to the log.

Table 8. Input link properties (continued)

Prompt	Type	Default	Description
Buffer Length	Long	4096	Required. Sets the length (in chunks greater than 512 bytes) of the FTP send and receive buffers for data rows before they are sent or retrieved.

You can specify any UNIX command for the following link properties: Telnet After Command, Telnet Backup Command, Transaction Begin Command, Transaction Commit Command, or Transaction Rollback Command. For example, the following UNIX command copies a file to another file in a different directory:

```
cp /pathname/filename1 /pathname2/filename2
```

Output Link Properties

The following table lists the output link properties in the grid editor:

Table 9. Output link properties

Prompt	Type	Default	Description
Remote Path	String	None	Optional. The path name of the working directory on the remote machine where the files to be retrieved or sent reside.
Remote File Name	String	None	Required. The name of the file on the remote machine to be retrieved or sent.
Data Representation Type	List	ASCII	Required. Controls how the data in the remote file is read or written. For **ASCII** representation, the data transfer uses standard NVT-ASCII, primarily for text files. For **binary** representation, the data is transferred in contiguous bits as IMAGE data. You must set "Fixed-width Columns" to Yes. **Note:** Binary mode is not supported when the stage is run on the Parallel canvas. To transfer data in binary mode, use data types of binary or varbinary with "Data Representation Type" set to ASCII.

Table 9. Output link properties (continued)

Prompt	Type	Default	Description
Check Data against metadata	List	No	Set to Yes to use metadata definitions to read data from the remote file instead of using a line terminator to identify the end of a row. Data is read until the metadata is exhausted. For fixed-width data, this means the total of the column lengths plus spaces. For delimited data, this means the number of columns. If set to No, end of row is determined by the end-of-line sequence [CR] [LF]
Line Termination	List	[CR] [LF] (DOS-Style Termination)	Specifies the row (end-of-line) termination sequence in the remote file. If "Fixed-width Columns" is set to No, use the [CR] [LF] value. If "Fixed-width Columns" is set to Yes, and "Data Representation Type" is set to ASCII, the valid values are no termination and [CR] [LF] (DOS style terminator). If set to no termination, "Check Data against metadata" must be set to Yes.
Fixed-width Columns	String	No	Required. Indicates whether the data in the remote file can be with fixed-width columns.
Spaces Between Columns	Long	0	The number of spaces between fixed-width columns in the remote file. Required if "Fixed-width Columns" is set to Yes.
Column Delimiter	Char	, (comma)	Required if "Fixed-width Columns" is set to No. The delimiter that separates the data fields in the remote file. You can enter single character without quotes or the ASCII value of the character you want to use.

Table 9. Output link properties (continued)

Prompt	Type	Default	Description
Quote Character	Char	" (double quote)	Optional and only valid if "Fixed-width Columns" is set to No. The single character used to enclose a data value that contains the delimiter character as data. You can also enter the ASCII value for the character you want to use. You can suppress "Quote Character" by entering no value.
Escape Character	Char	\ (back- slash)	Required. The single character entered to be interpreted as the escape character.
Null String	String	None	Optional. Specifies the string that is to be interpreted as the SQL null value.
First Line Column Names	String	No	Required if "Data Representation Type" is set to ASCII. Specifies whether to transfer the first line in the remote file (that is, it might contain column names).
Telnet Before Command	String	None	Optional. The telnet command to execute on the remote machine before starting a job.
Telnet After Command	String	None	Optional. Specifies the telnet command to execute on the remote machine after completing a job.
FTP Data Connection Mode	List	Active	Specifies which process initiates the connection for the data transfer. If set to Active, connections are initiated by the FTP server. If set to Passive, connections are initiated from the host system where the engine tier is installed. This lets you store files on remote hosts that are outside router-based firewall.

Table 9. Output link properties (continued)

Prompt	Type	Default	Description
FTP Data Port	List	None	Optional. The unique post number on which to receive the data from the remote machine's FTP server. The remote machine's FTP server connects to this port to transfer the remote file. If you do not specify a value, or the value is 0, the stage automatically configures an available port number for you. If you specify a value, it must be from 1025 to 4999. For more information on the FTP model, see the standard, RFC 959 File Transfer Protocol (FTP).

Link Tracing Level	Long	0	Optional. Controls the type of tracing information that is added to the log. The available tracing levels are:
			`0 No tracing`
			`1 Link properties`
			`2 Performance`
			`4 FTP messages`
			`8 Telnet messages`
			`16 Function tracing`
			`32 Telnet data dump`
			
			You can combine the tracing levels. For example, a tracing level of 3 means that link properties and performance messages are added to the log.

Buffer Length	Long	4096	Required. Sets the length (in chunks greater than 512 bytes) of the FTP send and receive buffers for data rows before they are sent or retrieved.

ASCII or Binary Data Representation

ASCII data representation for output links. If you set **Data Representation Type** to **ASCII**:

- The FTP service is configured for ASCII representation type. The sender (remote host) converts the data from its internal character representation, that is, ASCII or EBCDIC, to the standard NVT-ASCII representation. For more information on FTP data representation and storage, see the standard, RFC 959 File Transfer Protocol (FTP).
- The data stream received from the remote host is parsed into rows of data by scanning the data for the end-of-line sequence [CR] [LF].

- The row of data is further parsed into column data. The parsing method used depends on the setting for **Fixed-width Columns**. If set to **Yes**, the column metadata determines field sizes. If set to **No**, the row is parsed into columns by scanning for the column delimiter.

ASCII data representation for input links. If you set **Data Representation Type** to **ASCII**:

- The FTP service is configured for ASCII representation type. The receiver (remote host) converts the data from ASCII format to its own internal format.
- Column data (per row) is put in a formatted row. The format depends on the setting for **Fixed-width Columns**. If set to **Yes**, the data is put in a character buffer. The column metadata determines the size allotted per column. If the column data is greater than the column width, the data is truncated to the metadata column-width, and a warning message appears. If set to **No**, the data is put in a character buffer, separated by the delimiter for the configured column.
- The termination characters [CR] [LF] are appended to each row of data. The data is sent to the remote machine to be stored as a text file.

Binary data representation for output links. If you set **Data Representation Type** to **Binary**:

- **Fixed-width Columns** must be set to **Yes**.
- The FTP service is configured for IMAGE representation type. The data is sent from the remote machine as contiguous bits with no character conversions.
- The data steam received from the remote machine is parsed into rows of data by determining the total length of the row. The row length is calculated by the accumulation of each column's width and the values associated with **Spaces Between Columns** and **Line Termination**.
- The row of data is further parsed into column data using the same properties and metadata.

  Note: Binary mode is not supported on the Parallel Server canvas. See the input property Data Representation Type and the output property Data Representation Type.

Binary data representation for input links. If you set **Data Representation Type** to **Binary**:

- **Fixed-width Columns** must be set to **Yes**.
- The FTP service is configured for IMAGE representation type. The data is sent as contiguous bits with no character conversions.
- Column data per row is put in a character buffer. The column metadata determines the size allotted per column. If the column data is greater than the column width, the data is truncated to the metadata column-width, and a warning message appears.
- The termination characters specified by **Line Termination** are appended to each row of data and are sent to the remote machine.

  Note: Binary mode is not supported on the Parallel Server canvas. See the input property Data Representation Type and the output property Data Representation Type.

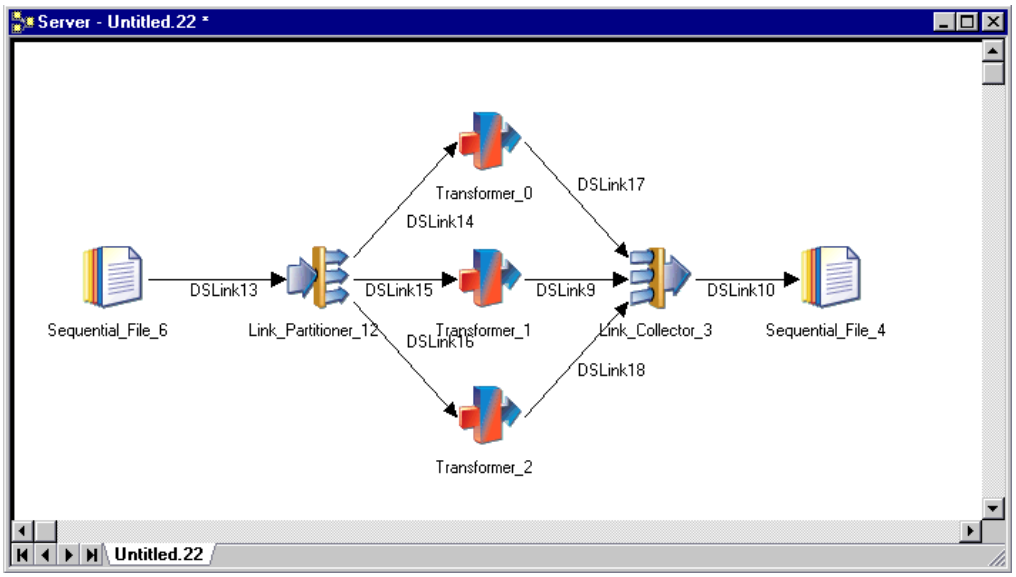
Link Collector Stages

These topics describe how to use a Link Collector stage in your job design.

The Link Collector stage is an active stage which takes up to 64 inputs and allows you to collect data from these links and route it along a single output link. The stage expects the output link to use the same metadata as the input links.

The Link Collector stage can be used with a Link Partitioner stage to enable you to take advantage of a multiprocessor system and have data processed in parallel. The Link Partitioner stage partitions data, it is processed in parallel, then the Link Collector stage collects it together again before writing it to a single target. To really understand the benefits, see “IBM InfoSphere DataStage Jobs and Processes” on page 5 to learn how IBM InfoSphere DataStage jobs are run as processes.

The following diagram illustrates how the Link Collector stage can be used in a job in this way:



In order for this job to compile and run as intended on a multiprocessor system you must have interprocess buffering turned on, either at project level using the Administrator client, or at the job level from the **Job Properties** dialog box.

The temporary files generated by this stage are placed in the directory specified by the TEMP environment variable. Use the Administrator client to set TEMP on a per-project basis.

Using a Link Collector Stage

When you edit a Link Collector stage, the Link Collector Stage dialog box appears. This dialog box has three pages:

- **Stage**. Displays the name of the stage you are editing. This page has a **General** tab which contains an optional description of the stage and the names of before- and after-stage routines. For more details about these routines, see "Before-Stage and After-Stage Subroutines." It also has a **Properties** tab that allows you to specify properties which affect the way the stage behaves. For details see "Defining Link Collector Stage Properties" on page 77.
- **Inputs**. Specifies the column definitions for the data input links.
- **Outputs**. Specifies the column definitions for the data output link.

Click **OK** to close this dialog box. Changes are saved when you save the job.

Before-Stage and After-Stage Subroutines

The **General** tab on the Stage page contains optional fields that allow you to define routines to use, which are executed before or after the stage has processed the data.

- **Before-stage subroutine** and **Input Value**. Contain the name (and value) of a subroutine that is executed before the stage starts to process any data. For example, you can specify a routine that prepares the data before processing starts.
- **After-stage subroutine** and **Input Value**. Contain the name (and value) of a subroutine that is executed after the stage has processed the data. For example, you can specify a routine that sends an electronic message when the stage has finished.

Choose a routine from the list. This list contains all the routines defined as a **Before/After Subroutine** in the **Routines** folder in the repository tree. Enter an appropriate value for the routine's input argument in the **Input Value** field.

If you choose a routine that is defined in the repository, but which was edited but not compiled, a warning message reminds you to compile the routine when you close the **Link Collector Stage** dialog box.

A return code of 0 from the routine indicates success, any other code indicates failure and causes a fatal error when the job is run.

If you installed or imported a job, the **Before-stage subroutine** or **After-stage subroutine** field might reference a routine that does not exist on your system. In this case, a warning message appears when you close the Link Collector Stage dialog box. You must install or import the "missing" routine or choose an alternative one to use.

Defining Link Collector Stage Properties

The **Properties** tab allows you to specify two properties for the Link Collector stage:

- **Collection Algorithm**. Use this property to specify the method the stage uses to collect data. Choose from:
  - **Round-Robin**. This is the default method. Using the round-robin method, the stage will read a row from each input link in turn.
  - **Sort/Merge**. Using the sort/merge method, the stage reads multiple sorted inputs and writes one sorted output.
- **Sort Key**. This property is only significant where you have chosen a collecting algorithm of Sort/Merge. It defines how each of the partitioned data sets are known to be sorted and how the merged output will be sorted. The key has the following format:

  Columnname {*sortorder*] [,*Columnname* [*sortorder*]]...

  Columnname specifies one (or more) columns to sort on.

  sortorder defines the sort order as follows:

Ascending Order	Descending Order
a	d
asc	dsc
ascending	descending
A	D
ASC	DSC
ASCENDING	DESCENDING

  In an NLS environment, the collate convention of the locale might affect the sort order. The default collate convention is set in the Administrator client, but can be set for individual jobs in the **Job Properties** dialog box.

  For example:

  ```
  FIRSTNAME d, SURNAME D
  ```

  Specifies that rows are sorted according to FIRSTNAME column and SURNAME column in descending order.

Defining Link Collector Stage Input Data

The Link Collector stage can have up to 64 input links. This is where the data to be collected arrives. The **Input name** list on the Inputs page allows you to select which of the 64 links you are looking at.

The Inputs page has two tabs:

- **General**. The **General** tab allows you to specify an optional description of the stage.
- **Columns**. The **Columns** tab contains the column definitions for the data on the input links. This is normally populated by the metadata of the stages connecting on the input side. You can also **Load** a column definition from the repository, or type one in yourself (and **Save** it to the repository if required). Note that the metadata on all input links must be identical, and this in turn must be identical to the metadata on the output link.

Defining Link Collector Stage Output Data

The Link Collector stage can have a single output link.

The Outputs page has two tabs: **General** and **Columns**.

- **General**. The **General** tab allows you to specify an optional description of the stage.
- **Columns**. The **Columns** tab contains the column definitions for the data on the output link. You can **Load** a column definition from the repository, or type one in yourself (and **Save** it to the repository if required). Note that the metadata on the output link must be identical to the metadata on the input links.

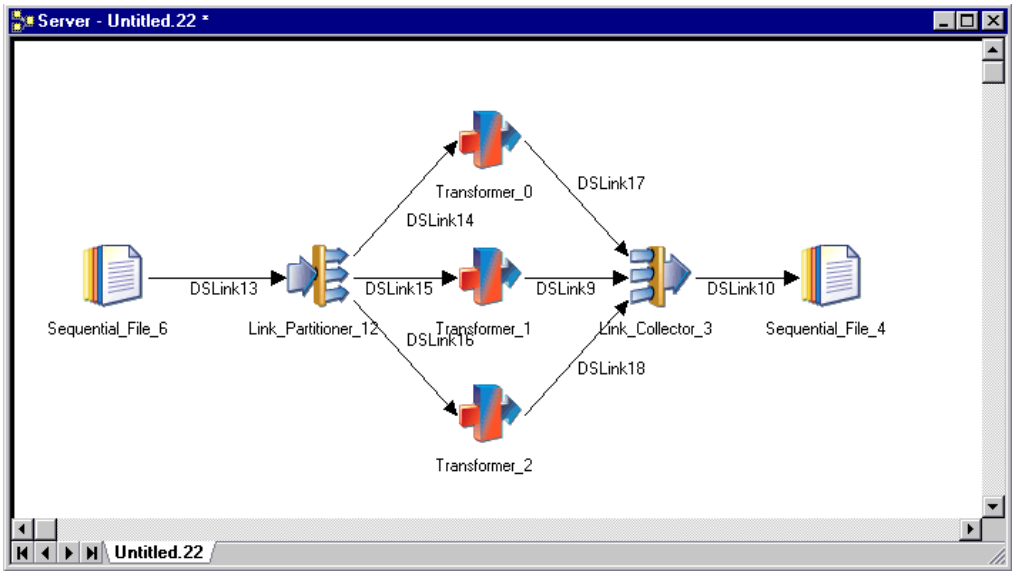
Link Partitioner Stages

These topics describe how to use a Link Partitioner stage in your job design.

The Link Partitioner stage is an active stage which takes one input and allows you to distribute partitioned rows to up to 64 output links. The stage expects the output links to use the same metadata as the input link.

Partitioning your data enables you to take advantage of a multiprocessor system and have the data processed in parallel. It can be used with the Link Collector stage to partition data, process it in parallel, then collect it together again before writing it to a single target. To really understand the benefits, see “IBM InfoSphere DataStage Jobs and Processes” on page 5 to learn how IBM InfoSphere DataStage jobs are run as processes.

The following diagram illustrates how the Link Partitioner stage can be used in a job in this way.



In order for this job to compile and run as intended on a multiprocessor system you must have interprocess buffering turned on, either at project level using the Administrator client, or at the job level from the **Job Properties** dialog box.

The temporary files generated by this stage are placed in the directory specified by the TEMP environment variable. Use the Administrator client to set TEMP on a per-project basis.

Using a Link Partitioner Stage

When you edit a Link Partitioner stage, the Link Partitioner Stage dialog box appears. This dialog box has three pages:

- **Stage**. Displays the name of the stage you are editing. This page has a **General** tab which contains an optional description of the stage and names of before- and after-stage routines. For more details about these routines, see "Before-Stage and After-Stage Subroutines." It also has a **Properties** tab that allows you to specify properties which affect the way the stage behaves. For details see "Defining Link Partitioner Stage Properties" on page 80.
- **Inputs**. Specifies the column definitions for the data input link.
- **Outputs**. Specifies the column definitions for the data output links.

Click **OK** to close this dialog box. Changes are saved when you save the job.

Before-Stage and After-Stage Subroutines

The **General** tab on the Stage page contains optional fields that allow you to define routines to use which are executed before or after the stage has processed the data.

- **Before-stage subroutine** and **Input Value**. Contain the name (and value) of a subroutine that is executed before the stage starts to process any data. For example, you can specify a routine that prepares the data before processing starts.
- **After-stage subroutine** and **Input Value**. Contain the name (and value) of a subroutine that is executed after the stage has processed the data. For example, you can specify a routine that sends an electronic message when the stage has finished.

Choose a routine from the list. This list contains all the routines defined as a **Before/After Subroutine** in the **Routines** folder in the repository tree. Enter an appropriate value for the routine's input argument in the **Input Value** field.

If you choose a routine that is defined in the repository, but which was edited but not compiled, a warning message reminds you to compile the routine when you close the **Link Partitioner Stage** dialog box.

A return code of 0 from the routine indicates success, any other code indicates failure and causes a fatal error when the job is run.

If you installed or imported a job, the **Before-stage subroutine** or **After-stage subroutine** field might reference a routine that does not exist on your system. In this case, a warning message appears when you close the Link Partitioner Stage dialog box. You must install or import the "missing" routine or choose an alternative one to use.

Defining Link Partitioner Stage Properties

The **Properties** tab allows you to specify two properties for the Link Partitioner stage:

- **Partitioning Algorithm**. Use this property to specify the method the stage uses to partition data. Choose from:
  - **Round-Robin**. This is the default method. Using the round-robin method, the stage will write each incoming row to one of its output links in turn.
  - **Random**. Using this method, the stage will use a random number generator to distribute incoming rows evenly across all output links.
  - **Hash**. Using this method, the stage applies a hash function to one or more input column values to determine which output link the row is passed to.
  - **Modulus**. Using this method, the stage applies a modulus function to an integer input column value to determine which output link the row is passed to.
- **Partitioning Key**. This property is only significant where you have chosen a partitioning algorithm of **Hash** or **Modulus**. For the **Hash** algorithm, specify one or more column names separated by commas. These keys are concatenated and a hash function applied to determine the destination output link. For the **Modulus** algorithm, specify a single column name which identifies an integer numeric column. The value of this column value determines the destination output link.

Defining Link Partitioner Stage Input Data

The Link Partitioner stage can have one input link. This is where the data to be partitioned arrives.

The Inputs page has two tabs:

- **General**. The **General** tab allows you to specify an optional description of the stage.
- **Columns**. The **Columns** tab contains the column definitions for the data on the input link. This is normally populated by the metadata of the stage connecting on the input side. You can also **Load** a column definition from the repository, or type one in yourself (and **Save** it to the repository if required). Note that the metadata on the input link must be identical to the metadata on the output links.

Defining Link Partitioner Stage Output Data

The Link Partitioner stage can have up to 64 output links. Partitioned data flows along these links. The **Output name** list on the Outputs page allows you to select which of the 64 links you are looking at.

The Outputs page has two tabs:

- **General**. The **General** tab allows you to specify an optional description of the stage.
- **Columns**. The **Columns** tab contains the column definitions for the data on the output link. You can **Load** a column definition from the repository, or type one in yourself (and **Save** it to the repository if

required). Note that the metadata on the output link must be identical to the metadata on the input link. Thus the metadata is identical for all of the output links.

Merge Stages

The Merge stage allows you to combine two sequential files into one or more output links. Merge, a passive stage, has no input links but has at least one output link. Use the graphical user interface (GUI) to define the join operation that is used to merge two files. The two input files to be merged must be sequential text files.

Functionality

The Merge stage supports the following functionality:

- Combining two sequential text files.
- Choosing from among several different types of joins.
- Support for NLS (National Language Support) in automatic mode only.

Using the Merge Stage

The following is a series of tasks required to merge two files, listed in the order in which you might perform them. You must specify:

- The input file and working file directories and log file information. Refer to "The General Tab of the Stage Page."
- The file names of the files to be merged. Refer to The General Tab.
- The type of join. Refer to Join Type.
- The tracing level. Refer to Tracing Level.
- The input file format. Refer to "The Input File Properties Tab" on page 84.
- Input file columns. Refer to First and Second File Columns Tabs.
- Where to save column information. Refer to The Save Table Definition Dialog Box.
- Keys for the join. Refer to "The Mapping Tab" on page 86.
- The content of the output columns. Refer to "Specifying Output Columns" on page 86.
- The name and format of the output file columns. Refer to "The Columns Tab" on page 87.

The General Tab of the Stage Page

The General tab of the **Stage** page defines input file and working file directories and log file information. The **General** tab contains the following fields:

- **First File Directory Path**. The path and directory of the first sequential file.
- **... (Browse Button).** Clicking **...** opens the Select from Server dialog box. See "Select from Server Dialog Box" on page 82.
- **Second File Directory Path.** The path and directory of the second sequential file.
- **... (Browse Button).** Clicking **...** opens the Select from Server dialog box. See "Select from Server Dialog Box" on page 82.
- **Temporary Directory.** The complete path and directory in which temporary files are stored. These temporary files are created while a job is running and deleted when the job is complete. The default is the current working directory.
- **... (Browse Button).** Clicking **...** opens the Select from Server dialog box. See "Select from Server Dialog Box" on page 82.
- **Tracing Level.** The type of information to be included in the job log file. You can specify the following tracing levels:

- 0 - No information is written to the log file
- 1 - Stage properties are written to the log file

  The default is 0.

- **Description.** An optional description of the stage properties.

Note: You can also include a job parameter in the directory path.

Select from Server Dialog Box

If you click Browse (**...** button), the Select from Server dialog box opens. The following fields are in the Select from Server dialog box:

- **Look in.** The name of the default directory selected. Click the down arrow to see where in the directory hierarchy you are currently located.
- **Directory or file names.** A list of names of directories or files under the directory in **Look in**.
- **File name (or pattern).** The name of the selected file or pattern.
- **Files of type.** The file name extension. By default, all files are displayed.

Defining Character Set Mapping

You can define a character set map for a stage. Do this from the **NLS** tab on the Stage page. The **NLS** tab appears only if you have installed NLS.

Specify information using the following fields:

- **Map name to use with stage.** Defines the default character set map for the project or the job. You can change the map by selecting a map name from the list.
- **Show all maps.** Lists all the maps that are shipped with InfoSphere DataStage.
- **Loaded maps only.** Lists only the maps that are currently loaded.
- **Allow per-column mapping.** Enable character set mapping on a column basis. Columns within a record can use different maps within the metadata.
- **Use Job Parameter....** Specifies parameter values for the job. Use the format #*Param*#, where *Param* is the name of the job parameter. The string #*Param*# is replaced by the job parameter when the job is run.

Adjusting for Input File Size

The Merge stage supports 64-bit files. But you must change the value of the property **Max Space in VM for Hash Table** to accommodate extremely large input files. Failure to do so results in abnormal termination of jobs. The default value of **Max Space in VM for Hash Table** is 12. This value is appropriate for many file sizes. As the size of the larger of the two input files grows, you must increase the value of **Max Space in VM for Hash Table**. For files of 2 GB or larger, you must set the value of **Max Space in VM for Hash Table** to its maximum value of 512.

To access **Max Space in VM for Hash Table**, right-click the **Merge** icon on the canvas, and select **Grid Style**. The grid-style editor appears. Go to the **Properties** tab of the Output page. Scroll the list of properties until you come to **Max Space in VM for Hash Table**.

Defining Output Properties

The Output page in theMERGE stage dialog box lets you specify properties for the output link. Output properties describe different characteristics of your input files and the output link, such as the following:

- Names of the first and second input files

- Output link tracing level
- Format of the first and second input files
- Column names and characteristics of the first and second input files, including character set mapping
- Column information to be saved to a table
- Type of join operation to be performed
- Keys used in the join operation
- Content of columns in the output link
- Column names and formats in the output link, including character set mapping

The General Tab

When you select the Output page, the **General** tab opens.

Note: The **Columns...** button lists the columns in the output link and is included only for compatibility with other stages.

The **General** tab contains the following fields:

- **First File Name.** The directory path and file name of the first file to be merged. This file must be a sequential text file. You can also include a job parameter in the directory path.
- **Second File Name.** The directory path and file name of the second file to be merged. This file must be a sequential text file. You can also include a job parameter in the directory path.
- **Join Type.** The type of join you want to perform on the two input files. You can choose one of the following types of join operations:

Type of Join	Operation	Description
Pure Inner Join	A AND B	Merges only those rows with the same key values in both input files.
Complete Set	A OR B	Merges all rows from both files.
Right and Left Only	A NOR B	Merges all rows from both files except those rows with the same key values.
Left Outer Join	A	Merges all rows from the first file (A) with rows from the second file (B) with the same key value.
Right Outer Join	B	Merges all rows from the second file (B) with rows from the first file (A) with the same key value.
Left Only	A NOT B	Merges all rows from the first file except rows with the same key value in the second file (B).
Right Only	B NOT A	Merges all rows from the second file except rows with the same key value in the first file (A).

- **Tracing Level.** Specifies a tracing level for the output link. The tracing level specifies the type of information to be included in the job log file. You can specify the following tracing levels:
  - 0 - No information is written to the log file
  - 1 - Output link properties are written to the log file

The Input File Properties Tab

You must specify the file format of the first and second input files. To specify the file format, click the **Input Files Properties** tab on the Output page. The **First** File Format page opens at the front of the Input Files Properties page.

To specify the format of the second input file, click the **Second File Format** tab. The fields and check boxes are identical for the second file. The following describes each field and check box on the **First File Format** or **Second File Format** pages:

First and Second File Format Tabs:

- **Fixed-width columns.** Indicates whether the file has fixed-width columns. The default is cleared.
- **First line is column names.** Indicates whether the first line of the first sequential file is column names. The default is cleared.
- **Check data against metadata.** Indicates whether to use metadata definitions to read data from the file instead of using a line terminator for the end of a row. Data is read until the metadata is exhausted.
  - For fixed-width data, this means the total of the column lengths plus spaces.
  - For delimited data, this means the number of columns.
  - If cleared, the end of row is determined by the end-of-line sequence.

    The default is cleared.
- **Delimiter.** Specifies the delimiter that separates the data fields in the file. This option is enabled if **Fixed-width columns** is cleared. You can enter an unquoted single character or the ASCII value of the character you want to use. The default is **,** (comma).
- **Quote character.** Specifies the character used to enclose a data value that contains the delimiter character as data. This option is enabled if **Fixed-width columns** is cleared. You can also enter the three digit ASCII value for the character you want to use. All values of length 1 to 2 will be treated as strings. You can enter '097' for 'a'. You can suppress **Quote character** by not entering a value. The default is **"** (double quotation marks).
- **Escape character.** Specifies a single character to be interpreted as an escape character. This option is enabled if **Fixed-width columns** is cleared. The default is \ (backslash).
- **Spaces between columns.** Specifies the number of spaces between columns in a sequential file with fixed-width columns. The default value is 0.
- **NULL string.** Specifies the string used for the SQL null value. There is no default.
- **Unix Style (LF).** Specifies whether a line-feed character is used to indicate the end-of-line sequence in the input file. The default is **Unix Style (LF)** not selected.
- **Dos Style (CR LF).** Specifies whether a combination of carriage-return and line-feed characters is used to indicate the end-of-line sequence in the input file. The default is **DOS Style (CR LF)** not selected.
- **None.** Specifies whether to use an end-of-line terminator. **None** is enabled if **Fixed-width columns** and **Check data against metadata** are selected. The default is **None** not selected.

First and Second File Columns Tabs:

Using the **First File Columns** and Second File Columns pages, you can specify the following:

- Column names of the first and second sequential input files
- Sequential file characteristics, including SQL type, length, scale, nullable, and display of the column
- Character set map used for the column

Click the **First** (or **Second**) **File Columns** tab on the Input Files Properties page. The **First** (or **Second**) File Columns page opens.

You have two options when entering information about the columns:

- You can use information from an existing table to specify the input file columns.
- You can enter the column information manually.

Using Column Information from an Existing Table:

You can use information from an existing table to define the columns in the first and second input files. Table definitions specify the data used at each stage of an InfoSphere DataStage job and are stored in the repository.

To transfer information about columns from an existing table:

1. Click **Load...** . The Table Definition dialog box appears.
2. Use the mouse to select the table definition in the left pane and click **OK**. The listed tables are already defined in the repository.
   a. If you don't know the table definition, click **Find...** . The Find dialog box appears.
   b. In the **Find what** field, enter a text string. The first table definition that contains the text string you specify is highlighted in the left pane.
3. Once you select the file name, click **OK**.

Entering Column Information Manually:

You can enter information about columns manually, by entering the information on the First File Columns page.

Enter a column name in the **Column Name List** and use the **Column Actions** buttons (**Add**, **Insert Before**, **Modify**, **Remove**, or **Remove All)** to specify where to put the names in the **Column Name List**.

You are then prompted to enter the information described next.

- **Column Name List.** Specifies the names of each column in either the first or second files. These names are used in the Mapping page that defines the output link. There is no default.
- **SQL Type.** Specifies the SQL data type. There is no default.
- **Length.** Defines the data precision. It is the length for CHAR data or the maximum length for VARCHAR data. For numeric data, it is the number of digits of precision. The default value is **0**.
- **Scale.** Specifies the data scale factor. For numeric data, it is the number of digits to the right of the decimal point. The default value is **0**.
- **Nullable.** Specifies whether the column can contain null values. The default value is **Yes**.
- **Display.** Specifies the maximum number of characters required to display the column data. The default value is **0**.
- **NLS map**. Specifies a different mapping for the column if **per-column mapping** is enabled (see "Defining Character Set Mapping" on page 82). Select a map from the list.

The Save Table Definition Dialog Box:

Once you have specified the column names and corresponding information the way you want, you can write that information to a new table. To save column information in a table, Click **Save...**. The Save Table Definition dialog box opens. The Save Table Definition dialog box contains the following fields;

- **Data source type.** The type of data written to the table. The data source type can be an ODBC data source, a UniVerse table, a hashed (UniVerse) file, a UniData file, a sequential file, or a stage. The table definition is stored according to the data source in the **Table Definitions** branch. The default is **Saved**.
- **Data source name.** Forms the second part of the table definition identifier and provides the name of the branch created under the data source type. It provides a means to track where the data definition originated. The default is the link name.
- **Table/file name.** The table or file name containing the data. The default is the link name.

- **Short description.** An optional brief description of the data. The default is the time and date saved.
- **Long description.** An optional long description of the data.

The Mapping Tab

You must specify the keys in the first and second sequential input files to be used in the join operation. To specify the keys, click the **Mapping** tab on the Output page. The **Mapping** tab opens.

Specifying Keys for the Join:

Select the keys from **First** (and **Second**) **File Column Names** on the left side of the page and drag them over to **First** (and **Second**) **File Column Key** on the right. These keys are used in the join operation to compare the two files.

You can specify multiple keys for the join operation. If you use multiple keys, you must have the same number of keys in the **First File Column Key** and **Second File Column Key** lists.

To delete an entry you made, select it and then right-click and choose **Clear Entry** from the shortcut menu.

Specifying Output Columns:

You must specify the contents of the columns to be included in the output link. Use the Mapping page to specify the contents of these columns.

In the Mapping page, the **First File Column Names** and **Second File Column Names** are already defined. You defined these in the Input Files Properties page.

In the Mapping page, you must specify which columns from the input files you want included in the output link. To specify the contents of a column in the output link, select a column from the **First File Column Names** or **Second File Column Names** list box and drag the column to the **Input Column Map** list. The **Output Column Name** is automatically generated. The properties of the columns in the output link are derived from those in the input file. You must include a **First File Column Key** and **Second File Column Key** in the **Column List**.

If you want to explicitly specify the names and properties of the columns in the output link, go to the Columns page as described in “The Columns Tab” on page 87.

You can select multiple columns at once to be dragged from the **First** (or **Second**) **File Column Names** list to the **Input Column Map** list. To select multiple columns, select the first column you want and hold down the **Ctrl** key until all the columns you want are highlighted. Or you can hold down the **Shift** key and click to select multiple columns.

You can right-click to delete any item from the **Input Column Map** list. To delete columns from the output link, click the **Columns** tab, and delete the columns as described in “The Columns Tab” on page 87.

Note: If you change the **First File** (or **Second File**) **Column Names** on the left side of the page, you might need to verify the mapping information (that is, the map keys and column list) on the right side of the page. If the column names on the right side of the page do not match those on the left, drag the correct column names from the left side to the right side.

The Columns Tab

You can use the **Columns** tab to specify the name and the format of the columns in the output link. You can also use the **Columns** tab to specify a different character set map for the column so that columns within a record can use different maps.

As described in First and Second File Columns Tabs you can use information from an existing table to specify the columns. Refer to that section for an explanation of how to use the **Load...** button to transfer information from a table.

Note: You must set all columns to "Nullable," except when the merge is set to **Pure Inner Join** as described in Join Type.

The **Columns** tab contains the following:

- **Column name.** Specifies the name of the column whose format you are defining.
- **Group.** Specifies whether you want to group by this column. The default is **No**.
- **Derivation.** Specifies that you want to summarize using this column.
- **Key.** Defines whether the column is a key.
- **SQL Type.** Specifies the SQL data type. The default is **(Unknown)**.
- **Length.** Defines the data precision. It is the length for CHAR data or the maximum length for VARCHAR data. For numeric data, it is the number of digits of precision.
- **Scale.** Specifies the data scale factor. For numeric data, it is the number of digits to the right of the decimal point.
- **Nullable.** Specifies whether the column can contain null values. Must be **Yes** unless you are performing a Pure Inner Join. The default is **No**.
- **Display.** Specifies the maximum number of characters required to display the column data.
- **Data element.** Specifies the type of data in the column.
- **Description.** Specifies an optional text description of the column.
- **NLS map.** If per-column mapping is enabled, specifies the mapping performed for the column. Select one of the map names from the drop-down list. The default is that of the project (MS1252).

Deleting Columns in the Output Link:

Using the Columns page, you can delete columns you defined in the output link. To delete columns in the output link:

1. Select the row you want to delete.
2. Press the **Delete** key.

Pivot Stages

Pivot, an active stage, maps sets of columns in an input table to a single column in an output table. This type of mapping is called pivoting.

This stage pivots horizontal data, that is, columns within a single row into many rows. It repeats a segment of data that is usually key-oriented for each column pivoted so that each output row contains a separate value.

An input column set can consist of one or more columns. The pivoting usually results in an output table that contains fewer columns but more rows than the original input table.

This stage has no stage or link properties. It merely maps input rows to output rows.

Functionality

Supported Functionality

The Pivot stage has the following functionality:

- Supports horizontal pivots.
- NLS (National Language Support).

Unsupported Functionality

The following functionality is not supported:

- Compatibility with IBM InfoSphere DataStage releases before 7.0.
- Vertical pivots, that is, mapping vertical data in many rows into a single row. (Vertical pivots group one or more columns and map these columns to many columns in the grouped row in an output table.)
- A custom user interface.

Pivoting Data

A horizontal pivot maps columns within a row into many rows, that is, it repeats a segment of data for each column pivoted. The data is usually key-oriented.

Use the **Derivation** field in the output link column grid to specify the pivots. An empty field indicates that there is an input column name with the same name as the output column. This input column is mapped to the corresponding output column.

Single Derivation

If the **Derivation** field for an output column lists a single column name, the input column having the same name as that specified in the **Derivation** field is mapped to this output column. Any column having a single derivation is treated as a key and is likewise projected to each output row that is derived from the single input row.

Multiple Derivations

When an output column is derived from more than one input column, that is, more than one input column name is listed in its **Derivation** field, an output table with more rows than the input table results. Each input column specified in the **Derivation** field for the output columns is mapped to the output column. A new row is created for each of the specified input columns.

Examples

The examples described in the following sections show a pivot on the first quarter sales data for a particular enterprise. These examples illustrate the concepts for a horizontal pivot.

Input Link Columns

The following example illustrates data input to the Pivot stage.

The **Columns** tab of the Inputs page contains three input columns with sales data: JAN_Sales, FEB_Sales, and MARCH_Sales. The columns are as follows:

Table 10. Input columns

Column name	SQL type	Length	Scale
CUSTID	Integer	10	

Table 10. Input columns (continued)

Column name	SQL type	Length	Scale
LNAME	VarChar	10	
JAN_Sales	Decimal	10	2
FEB_Sales	Decimal	10	2
MARCH_Sales	Decimal	10	2

Note: For any column, the data type documented in **SQL Type** must be the same as the data type in the source table.

The data for the source rows for the input columns looks like this:

Table 11. Input Source Rows

CUSTID	LNAME	JAN_Sales	FEB_Sales	MARCH_Sales
100	Smith	$1,234.00	$1,456.00	$1,578.00
101	Yamada	$1,245.00	$1,765.00	$1,934.00

Output Link Columns

The following example illustrates how to specify what data is output by the Pivot stage.

The output link **Columns** tab contains a Sales column derived from the three input columns: JAN_Sales, FEB_Sales, and MARCH_Sales. The columns are as follows:

Table 12. Output columns

Column name	Derivation	SQL type	Length	Scale
CUSTID		Integer	10	
Last_Name	LNAME	VarChar	10	
Sales	JAN_Sales, FEB_Sales, MARCH_Sales	Decimal	10	2

Note: For any column, the data type documented in **SQL Type** must be the same as the data type in the target table.

The output column that is derived from a single input column is a key value. The key value is repeated in each row that results from the corresponding input row.

The maximum number of output rows that result from a single input row is determined by the output column that is derived from the most input columns. The three output rows of sales data that result from each input row in this example are as follows:

Table 13. Output Target Rows

CUSTID	Last_Name	Sales
100	Smith	$1,234.00
100	Smith	$1,456.00
100	Smith	$1,578.00
101	Yamada	$1,245.00
101	Yamada	$1,765.00
101	Yamada	$1.934.00

If the pivot includes any derivations with fewer than the maximum number of output rows but more than one row, the output row contains a null value for each column where a derivation is not available.

As an example, assume the customer is required to make payments on his account twice a year, in June and December. The source data might look like this:

Table 14. Payments Example

CUSTID	LNAME	JAN_Sales	FEB_Sales	MARCH_Sales	JUN_Pay	DEC_Pay
100	Smith	$1,234.00	$1,456.00	$1,578.00	$6,298.00	$7,050.00
101	Yamada	$1,245.00	$1,765.00	$1,934.00	$7,290.00	$7,975.00

Suppose the output link contains an additional derivation for payments:

Table 15. Output columns with payments details

Column name	Derivation	SQL type	Length	Scale
CUSTID		Integer	10	
Last_Name	LNAME	VarChar	10	
Sales	JAN_Sales, FEB_Sales, MARCH_Sales	Decimal	10	2
Payments	JUNE_Pay, DEC pay			

The output data in the target rows after the pivot looks like this:

Table 16. Output Data in Target Rows After Pivot

CUSTID	LNAME	Sales	Payments
100	Smith	$1,234.00	$6,298.00
100	Smith	$1,456.00	$7,050.00
100	Smith	$1,578.00	null
101	Yamada	$1,245.00	$7,290.00
101	Yamada	$1,765.00	$7,975.00
101	Yamada	$1,934.00	null

Row Merger Stages

The Row Merger stage reads data one row at a time from an input link. It merges all the columns into a single string of a specified format. It then writes the string on a given column of the output link. The stage can have a single input link and a single output link.

In normal operation of the Row Merger stage, each input row with multiple columns results in an output row of a single column. The stage also offers concatenation facilities, however. These facilities allow you to concatenate the result of each input row into a single string which is output when the stage detects an end-of-data (EOD) or end-of-transmission (EOT) signal (that signifies no more input rows are expected).

Note: The Row Merger stage is similar to the server Sequential File stage. The difference is that, while the Sequential File stage writes to a file, the Row Merger stage outputs to a link.

Functionality

Supported Functionality

The Row Merger stage supports the following functionality:

- The ability to reads one row at a time, merge all the columns from a row into a single string of a specified format, and then write the string to a given column of the output link.
- Aggregation of multiple rows of data.
- NLS (National Language Support). The stage writes what it reads without interpretation or conversion.

Stage Page General Tab

The **General** tab of the Stage page gives access to the concatenation facilities of the Row Merger stage. The **General** tab contains the following fields:

- **Multiple Lines**. This determines whether the Row Merger stage concatenates input rows into a single output row, or whether it outputs each input row as a separate output row. Select **Multiple Lines** to have the rows concatenated. By default it is not selected.
- **Line Termination**. This setting is only available if you have chosen the **Multiple Lines** option to specify that the stage is concatenating input rows. It specifies the character(s) that will be placed as a delimiter between the concatenated rows when they are output in a single row. Choose from the following settings:
  - **Unix Style (LF).** Places a linefeed character as a delimiter between each merged row.
  - **DOS style (CR LF).** Places a carriage return character and a linefeed character as a delimiter between each merged row.
  - **None.** Does not place a delimiter between the merged rows.
- **Description**. Enter an optional description of the stage.

Input Page

The Input page contains various tabs that describe the rows of data being input to the Row Merger stage.

The **General** tab contains a description field that allows you to enter an optional description of the input link. The **Format** tab and **Columns** tab are described below.

Format Tab

Use this tab to specify how the data read in individual columns in each input row will be formatted before being output in a single column. The tab contains the following fields:

- **Fixed-width columns**. Select this check box to output the data in fixed-width format. The width of each field is taken from the SQL display size of the input columns (set in the **Display** column in the Columns grid on the Inputs page **Columns** tab). This option is cleared by default.
- **Suppress row truncation warnings**. This option is only available when you have selected **Fixed-width columns**. If the input rows contain more columns that you have defined on the **Columns** tab, you will normally receive warnings about overlong rows when the job is run. If you want to suppress these messages (for example, you might only be interested in the first three columns and happy to ignore the rest), select this check box.
- **Delimiter**. This option is not available if you have selected **Fixed-width columns**. It specifies the delimiter used to separate the data fields in the output data that have been derived from the input columns. By default this field contains a comma. You can enter a single printable character or a decimal or hexadecimal number to represent the ASCII code for the character you want to use. Valid ASCII codes are in the range 1 to 253. Decimal values 1 through 9 must be preceded with a zero. Hexadecimal values must be prefixed with &h. Enter 000 to suppress the delimiter.
- **Quote Character**. This option is not available if you have selected **Fixed-width columns**. Specifies the character used to enclose strings. By default this field contains a double quotation mark. You can enter a single printable character or a decimal or hexadecimal number to represent the ASCII code for the character you want to use. Valid ASCII codes are in the range 1 to 253. Decimal values 1 through 9 must be preceded with a zero. Hexadecimal values must be prefixed with &h. Enter 000 to suppress the quote character.

- **Spaces between columns**. This option is only available if you have selected **Fixed-width columns**. Contains a number to represent the number of spaces used between columns. By default this is 0.
- **Default NULL string**. Contains characters which, when encountered in an input row, are interpreted as the SQL null value.
- **Default padding**. This option is only available if you have selected **Fixed-width columns**. Contains the character used to pad missing columns. This is # by default, but can be set to another character here.

The **Format** tab also has a **Load** button. If you have table definitions that include format information, you can load the format details from these table definitions directly onto the Format page:

1. Click **Load**. The Load Table Definitions dialog box appears.
2. Browse for the table definition containing the format you want to load.
3. Click **OK**. The format details are loaded.

Columns Tab

The entries in the columns grid specify the format of the data being read from the input rows. The grid has the standard fields that all column definitions have.

Output Page

The Output page contains various tabs that describe the data being output by the Row Merger stage.

General Tab

The **General** tab identifies the column to contain the merged data. The **General** tab contains the following fields:

- **Name of the column to merge**. A list contains a list of the defined output columns for this stage. Choose the column that you want to output the merged data in.
- **Description**. An optional description of the output link.

Columns Tab

The entries in the columns grid specify the format of the data being written to the output link. The grid has the standard fields that all column definitions have. The Derivation field is not used.

At the least you must define a column to carry the merged data. You can also define additional columns to carry the data as input to the stage.

Row Splitter Stages

The Row Splitter stage reads data one row at a time from an input link. It splits the data fields contained in a string into a number of columns. It then writes the columns to the output link. The stage can have a single input link and a single output link.

In normal operation of the Row Splitter stage, each input string processed results in an output row of multiple columns. In some cases, however, a single input string can represent several rows of input data. In this case the stage can deconcatenate these into separate rows for output.

Note: The Row Splitter stage is similar to the server Sequential File stage. The difference is that, while the Sequential File stage reads from a file, the Row Splitter stage reads from a link.

Functionality

Supported Functionality

The Row Splitter stage supports the following functionality:

- Ability to read one row at a time, split the data fields contained in a string into a number of columns, and then write the columns to the output link.
- Generation of multiple output rows.
- NLS (National Language Support). The stage writes what it reads without interpretation or conversion.

Stage Page General Tab

The **General** tab of the Stage page gives access to the deconcatenation facilities of the Row Splitter stage. The **General** tab contains the following fields:

- **Multiple Lines**. This determines whether the Row Splitter stage deconcatenates the input string into separate output rows, or whether it outputs each input string as a separate output row. Select **Multiple Lines** to have the rows deconcatenated. By default it is not selected.
- **Line Termination**. This setting is only available if you have chosen the **Multiple Lines** option to specify that the stage is deconcatenating input rows. It specifies the character(s) that are placed as a delimiter between the concatenated rows, so the stage knows where to split them. Choose between:
  - **Unix Style (LF).** The delimiter is a linefeed character.
  - **DOS style (CR LF).** The delimiter is a carriage return character and a linefeed character.
  - **None.** There is no delimiter.
- **Description**. Enter an optional description of the stage.

Input Page

The Input page contains various tabs that describe the rows of data being input to the Row Splitter stage.

General Tab

Use the **General** tab to identify the name of the column that contains the string from which the stage extracts the columns. The **General** tab contains the following fields:

- **Name of the column to split**. A list contains a list of the defined input columns for this stage. Choose the column that carries the string from which the stage will extract the columns.
- **Description**. Enter an optional description of the input link.

Columns Tab

The entries in the columns grid specify the format of the data read from the input link. The grid has the standard fields that all column definitions have.

At the least you must define a column to carrying the data string which the stage is splitting. You can also define additional columns if required. Any columns that are defined here and on the Output page **Columns** tab will be passed straight through the stage.

Output Page

The Output page contains various tabs that describe the data being output by the Row Splitter stage.

The **General** tab contains a description field that allows you to enter an optional description of the input link. The **Format** tab and **Columns** tab are described below.

Format Tab

Use this tab to specify how the input string is formatted, so the stage can split the columns out. The tab contains the following fields:

- **Fixed-width columns**. Select this check box if the incoming data is in fixed-width format. The width of each field is taken from the SQL display size of the outputs columns (set in the **Display** column in the Columns grid on the Output page **Columns** tab). This option is cleared by default.
- **Suppress row truncation warnings**. If the input row contains more data fields to be split out into columns than you have defined on the **Columns** tab, you will normally receive warnings about overlong rows when the job is run. If you want to suppress these messages (for example, you might only be interested in the first three columns and happy to ignore the rest), select this check box.
- **Missing Columns Message**. If there fewer data fields in the input row than you have defined columns for them to be split into, this option allows you to specify what action to take:
  - **Fatal.** A fatal error is written to the job log and the job aborts (this is the default.)
  - **Warning.** A warning message is written to the job log, SQL nulls are writing to the extra columns and the job continues.
  - **None.** No action is taken. SQL nulls are writing to the extra columns and the job continues.
- **Delimiter**. This option is not available if you have selected **Fixed-width columns**. It specifies the delimiter used to separate the data fields in the input data string. By default this field contains a comma. You can enter a single printable character or a decimal or hexadecimal number to represent the ASCII code for the character you want to use. Valid ASCII codes are in the range 1 to 253. Decimal values 1 through 9 must be preceded with a zero. Hexadecimal values must be prefixed with &h. Enter 000 to suppress the delimiter.
- **Quote Character**. This option is not available if you have selected **Fixed-width columns**. Specifies the character used to enclose strings. By default this field contains a double quotation mark. You can enter a single printable character or a decimal or hexadecimal number to represent the ASCII code for the character you want to use. Valid ASCII codes are in the range 1 to 253. Decimal values 1 through 9 must be preceded with a zero. Hexadecimal values must be prefixed with &h. Enter 000 to suppress the quote character.
- **Spaces between columns**. This option is only available if you have selected **Fixed-width columns**. Contains a number to represent the number of spaces used between columns. By default this is 0.
- **Default NULL string**. Contains characters which, when encountered in an input row, are interpreted as the SQL null value (this can be overridden for individual column definitions in the **Columns** tab).
- **Default padding**. This option is only available if you have selected **Fixed-width columns**. Contains the character used to pad missing columns. This is # by default, but can be set to another character here.

The **Format** tab also has a **Load** button. If you have table definitions that include format information, you can load the format details from these table definitions directly onto the Format page:

1. Click **Load**. The Load Table Definitions dialog box appears.
2. Browse for the table definition containing the format you want to load.
3. Click **OK**. The format details are loaded.

Columns Tab

The entries in the columns grid specify the format of the data being written to the output link. The grid has the standard fields that all column definitions have.

The Derivation field is not used.

Sort Stages

Sort, an active stage, sorts a variety of data. It sorts small amounts of data efficiently in memory when there is enough main memory available. It sorts large amounts of data using temporary disk storage, not virtual memory swap space.

The model for the Sort stage is the UNIX *sort* command, as used in a shell pipeline. Input data rows to be sorted arrive as lines of ASCII characters read from the *stdin* stream. You use command line arguments to specify how to sort these rows. The resulting sorted rows are written as lines of ASCII characters to the *stdout* stream.

In InfoSphere DataStage, the Sort stage receives a stream of rows using a single input link. The rows are already separated into individual column values. The values for the stage properties and column attributes specify how to sort these rows. The resulting sorted rows are written as column values to a single output link.

The Sort stage must have one input and one output link. Considerations for the columns in the rows for the input and output links include the following:

- A single input stream link provides rows of data to be sorted. The column type of the input column must be convertible to the type of the output column.
- A single output stream link receives sorted rows of data. Output rows have the same column order as input columns. The names of output columns can differ from the names of input columns.

The output link data type for each column determines the type of comparison to perform:

- Numeric comparison for numbers
- Date and time
- Character string (left to right sort for strings and timestamps)

Functionality

Supported Functionality

The Sort stage has the following functionality and benefits:

- Supports NLS (National Language Support).
- Supports an option to sort per column using a collating sequence map.
- Supports an option to request a stable sort. A stable sort preserves the input order of rows that compare as equal.
- Logs messages to report nonfatal warnings that can impact loss of precision of sorted data.
- Supports performance tuning parameters for efficient sorting, thus limiting virtual memory use.

Unsupported Functionality

The following functionality is not supported:

- Bulk loading for stream input links
- Stored procedures

Configurable Properties

You can configure properties to improve performance for the Sort stage.

Max Rows in Virtual Memory Property

Max Rows in Virtual Memory lets you regulate the amount of data in virtual memory. By limiting the total number of rows to sort, the sort algorithm performs incremental sorts. This reduces the virtual memory usage and excessive page swapping that occurs when you have a large amount of input data associated with the input link.

This property is used when the number of rows within the input link exceeds the supplied value for this property. The sort algorithm sorts rows in multiples of this value and stores these sorted groups of rows in temporary files. These temporary files are then merged together for the final sort.

Max Open Files Property

Max Open Files limits the number of intermediate data files that are created when incremental sorts are performed. The processing of the data is controlled by the following:

- **Max Open Files**
- **Max Rows in Virtual Memory**
- The actual number of rows associated with the input link

Example

Assume that the input link contains 100,000 rows of data, and **Max Rows in Virtual Memory** is set to 10,000 rows.

The sort algorithm reads in the first 10,000 rows from the input link, performs an intermediate sort, then stores the sorted data to a temporary file. The algorithm continues to group 10,000-row chunks from the input link, storing the sorted results in unique temporary files, until one of the following conditions is met:

- All of the input data has been processed into temporary files. The total number of temporary files is less than the value specified in **Max Open Files**.

  After the intermediate sorts, the 10 temporary files are merged and sorted together, resulting in the final sort that is written to the output link.
- The number of temporary files equals the value specified in **Max Open Files**.

  If, for example, **Max Open Files** is set to 5, the first 50,000 rows are processed as five temporary files, with 10,000 rows each. These temporary files are merged together to form a new temporary file with 50,000 rows of sorted data. The algorithm grabs the next 10,000 rows from the input link and continues with the intermediate sorts. This algorithm continues recursively until all the data is processed.

  Note: If the values of these parameters are too restrictive, a high number of intermediate sorts results with constant file merging.

Sort Criteria

The Sort stage accumulates input rows in memory, limited by **Max Rows in Virtual Memory**. It sorts the accumulated rows, storing them in disk files, if necessary. (Small sort sets can be sorted in memory.) It merges these stored files and writes the rows to the output link.

You can enter the values listed in the following table to specify the order of rows, depending on case-sensitivity

Table 17. Sort criteria

Case-Sensitivity	Ascending Order	Descending Order
Sensitive	a asc ascending	d dsc descending

Table 17. Sort criteria (continued)

Case-Sensitivity	Ascending Order	Descending Order
Insensitive	A	D
	ASC	DSC
	ASCENDING	DESCENDING

:

The following example specifies to sort the resulting rows in case-sensitive ascending order on the input link REGION column. It uses an external map file named CSM in the C:\USER directory on the CUSTOMER column, and descending order on the SALE_PRICE column (see **Sort Specifications** in "Stage Properties").

```
REGION asc, CUSTOMER ASC C:\USER\CSM, SALE_PRICE DSC
```

Collating Sequence Maps

You can specify collating sequence map sorting per column. The format of the map accommodates character encoding, such as single-byte, double-byte, and variable number of bytes. You can specify a separate map file for each column to be sorted. The map file is used in sorting character string values in that column. The map does not affect the sorting of noncharacter-string values, that is, numeric, date, time, and timestamp values.

A collating sequence map is a comma-delimited file containing two columns. The left column is a single character code (in a single- or multibyte encoding, as appropriate). Use an escape character to enter delimiter characters and arbitrary byte values.

The right column is an integer value using ASCII characters for the decimal digits. The column contains the numeric weight used when comparing corresponding characters in two strings. The lower the number, the earlier it sorts. If two characters have identical weights, they compare as equals. Any character not in the map compares higher than any character in the map. For example, the following sequence map contains these comma-delimited columns:

a,3 b,3 c,3 d,5 g,6 e,1

You can, for example, provide a collating sequence map to specify the collating sequence for the French alphabet.

Stage Properties

The following table includes these column heads:

- **Prompt** is the text that the job designer sees in the stage editor user interface.
- **Default** is the text used if the job designer does not supply any value.
- **Description** describes the properties.

The Sort stage supports the following stage properties:

Table 18. Sort stage properties

Prompt	Default	Description
Sort Specifications	None	The criteria by which the ASCII characters in the rows read from the input link are sorted. See "Sort Criteria" for more information.

Table 18. Sort stage properties (continued)

Prompt	Default	Description
Max Rows in Virtual Memory	10,000	The maximum number of rows (from 2 to 50,000) that can be sorted in virtual memory. The smaller the row, the more rows that can be sorted.
Temporary Directory	None	The path name where the temporary files that are created during the sort are stored. If you do not specify a path name, the current working directory on the computer that hosts the engine tier is used.
Escape Character	\ (backslash)	The single character used in the collating sequence map files to specify control characters.
Tracing Level	0	Controls the type of tracing information that is added to the log. The available tracing levels are: 0 No tracing 1 Stage properties 2 Performance 4 Important events You can combine the tracing levels. For example, a tracing level of 3 means that stage properties and performance messages are added to the log.
Stable Sort	No	Indicates whether the sort is a stable sort. A stable sort preserves the order of the input rows that compare equal.
Column Separator	, (comma)	The single character separating the two columns in each line of the collating sequence map file.
Max Open Files	10	The maximum number of files that can be open simultaneously. The larger the value, the better the performance. When using one or more of these stage instances in the job, the total number of open files of all the stage instances must not exceed 20.

Transformer Stages

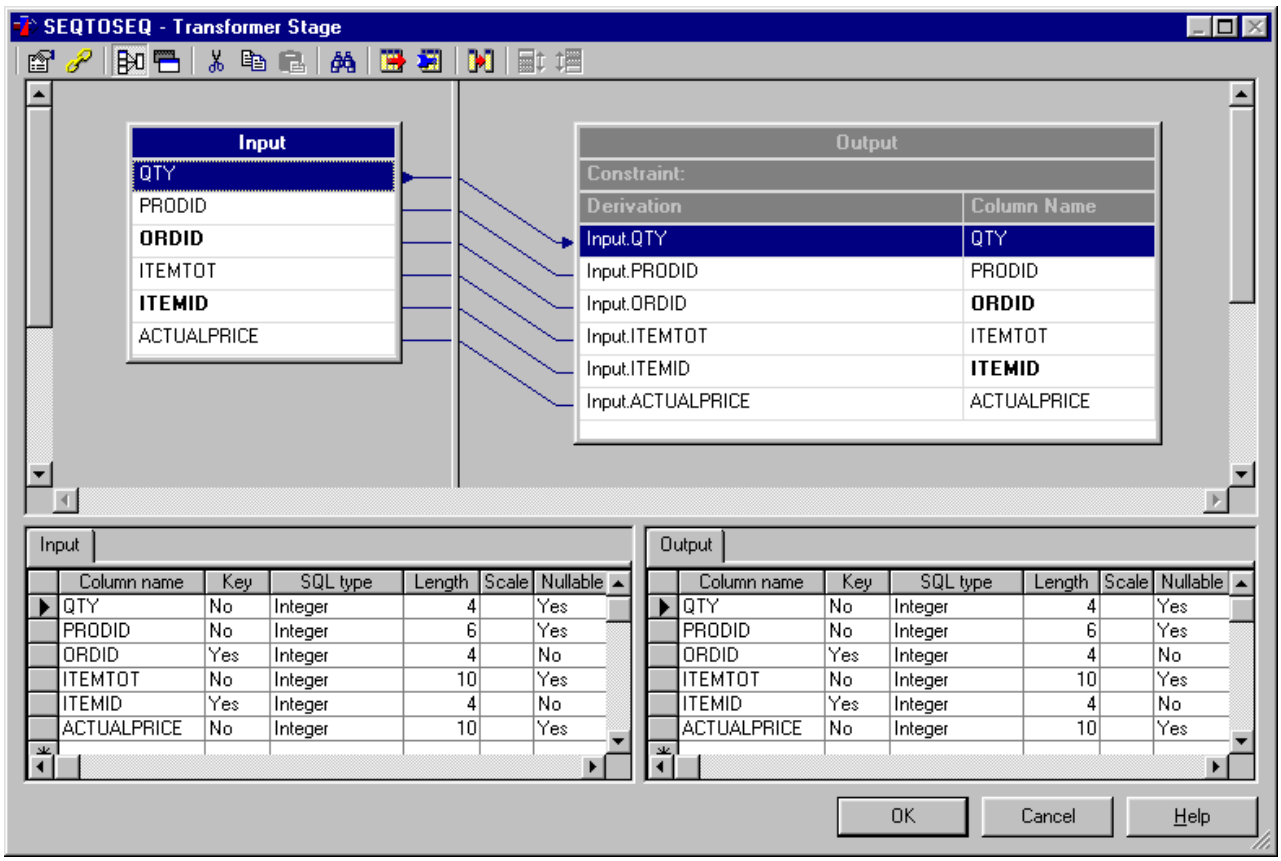
Transformer stages do not extract data or write data to a target database. They are used to handle extracted data, perform any conversions required, and pass data to another Transformer stage or a stage that writes data to a target data table.

Using a Transformer Stage

Transformer stages can have any number of inputs and outputs. The link from the main data input source is designated the primary input link. There can only be one primary input link, but there can be any number of reference inputs.

Note: The Transformer stage editor is similar for server, parallel, and mainframe jobs, but the functionality differs. Only the server job functionality is described in these topics. For parallel or mainframe job functionality, see the guides that describe parallel and mainframe jobs.

When you edit a Transformer stage, the Transformer Editor appears. An example Transformer stage is shown below. In this example, metadata has been defined for the input and the output links:



Transformer Editor Components

The Transformer Editor has the following components.

Toolbar

The Transformer toolbar contains the following buttons:

- Stage Properties
- Constraints
- Show All or Selected Relations
- Show/Hide Stage Variables
- Cut
- Copy
- Paste
- Find/Replace
- Load Column Definition
- Save Column Definition

- Column Auto-Match
- Input Link Execution Order
- Output Link Execution Order

Link Area

The top area displays links to and from the Transformer stage, showing their columns and the relationships between them.

The link area is where all column definitions, key expressions, and stage variables are defined.

The link area is divided into two panes; you can drag the splitter bar between them to resize the panes relative to one another. There is also a horizontal scroll bar, allowing you to scroll the view left or right.

The left pane shows input links, the right pane shows output links. The input link shown at the top of the left pane is always the primary link. Any subsequent links are reference links. For all types of link, key fields are shown in bold. Reference link key fields that have no expression defined are shown in red (or the color defined in **Tools → Options**), as are output columns that have no derivation defined.

Within the Transformer Editor, a single link can be selected at any one time. When selected, the link's title bar is highlighted, and arrowheads indicate any selected columns.

Metadata Area

The bottom area shows the column metadata for input and output links. Again this area is divided into two panes: the left showing input link metadata and the right showing output link metadata.

The metadata for each link is shown in a grid contained within a tabbed page. Click the tab to bring the required link to the front. That link is also selected in the link area.

If you select a link in the link area, its metadata tab is brought to the front automatically.

You can edit the grids to change the column metadata on any of the links. You can also add and delete metadata.

Shortcut Menus

The Transformer Editor shortcut menus are displayed by right-clicking the links in the links area.

There are slightly different menus, depending on whether you right-click an input link, an output link, or a stage variable. The input link menu offers you operations on key expressions, the output link menu offers you operations on derivations, and the stage variable menu offers you operations on stage variables.

The shortcut menu enables you to:

- Open the Properties dialog box to enter a description of the link.
- Open the Constraints dialog box to specify a constraint (only available for output links).
- Open the **Column Auto-Match** dialog box.
- Display the **Find/Replace** dialog box.
- Display the Select dialog box.
- Edit, validate, or clear a key expression, derivation, or stage variable.
- Edit several derivations in one operation.
- Append a new column or stage variable to the selected link.
- Select all columns on a link.

- Insert or delete columns or stage variables.
- Cut, copy, and paste a column or a key expression or a derivation or stage variable.

If you display the menu from the links area background, you can:

- Open the Stage Properties dialog box in order to specify a before- or after-stage subroutine.
- Open the Constraints dialog box in order to specify a constraint for the selected output link.
- Open the Link Execution Order dialog box in order to specify the order in which links should be processed.
- Toggle between viewing link relations for all links, or for the selected link only.
- Toggle between displaying stage variables and hiding them.

Right-clicking in the metadata area of the Transformer Editor opens the standard grid editing shortcut menus.

Transformer Stage Basic Concepts

When you first edit a Transformer stage, it is likely that you will have already defined what data is input to the stage on the input links. You will use the Transformer Editor to define the data that will be output by the stage and how it will be transformed. (You can define input data using the Transformer Editor if required.)

This section explains some of the basic concepts of using a Transformer stage.

Input Links

The main data source is joined to the Transformer stage via the primary link, but the stage can also have any number of reference input links.

A reference link represents a table lookup. These are used to provide information that might affect the way the data is changed, but do not supply the actual data to be changed.

Reference input columns can be designated as key fields. You can specify key expressions that are used to evaluate the key fields. The most common use for the key expression is to specify an equijoin, which is a link between a primary link column and a reference link column. For example, if your primary input data contains names and addresses, and a reference input contains names and phone numbers, the reference link **name** column is marked as a key field and the key expression refers to the primary link's **name** column. During processing, the name in the primary input is looked up in the reference input. If the names match, the reference data is consolidated with the primary data. If the names do not match, that is, there is no record in the reference input whose key matches the expression given, all the columns specified for the reference input are set to the null value.

Where a reference link originates from a UniVerse or ODBC stage, you can look up multiple rows from the reference table. The rows are specified by a foreign key, as opposed to a primary key used for a single-row lookup.

Output Links

You can have any number of output links from your Transformer stage.

You might want to pass some data straight through the Transformer stage unaltered, but it's likely that you'll want to transform data from some input columns before outputting it from the Transformer stage.

You can specify such an operation by entering a BASIC expression or by selecting a transform to apply to the data. IBM InfoSphere DataStage has many built-in transforms, or you can define your own custom transforms that are stored in the repository and can be reused as required.

The source of an output link column is defined in that column's **Derivation** cell within the Transformer Editor. You can use the Expression Editor to enter expressions or transforms in this cell. You can also simply drag an input column to an output column's **Derivation** cell, to pass the data straight through the Transformer stage.

In addition to specifying derivation details for individual output columns, you can also specify constraints that operate on entire output links. A constraint is a BASIC expression that specifies criteria that data must meet before it can be passed to the output link. You can also specify a reject link, which is an output link that carries all the data not output on other links, that is, columns that have not met the criteria.

Each output link is processed in turn. If the constraint expression evaluates to TRUE for an input row, the data row is output on that link. Conversely, if a constraint expression evaluates to FALSE for an input row, the data row is not output on that link.

Constraint expressions on different links are independent. If you have more than one output link, an input row might result in a data row being output from some, none, or all of the output links.

For example, if you consider the data that comes from a paint shop, it might include information about any number of different colors. If you want to separate the colors into different files, you would set up different constraints. You could output the information about green and blue paint on LinkA, red and yellow paint on LinkB, and black paint on LinkC.

When an input row contains information about yellow paint, the LinkA constraint expression evaluates to FALSE and the row is not output on LinkA. However, the input data does satisfy the constraint criterion for LinkB and the rows are output on LinkB.

If the input data contains information about white paint, this does not satisfy any constraint and the data row is not output on Links A, B or C, but will be output on the reject link. The reject link is used to route data to a table or file that is a "catch-all" for rows that are not output on any other link. The table or file containing these rejects is represented by another stage in the job design.

Before-Stage and After-Stage Routines

Because the Transformer stage is an active stage type, you can specify routines to be executed before or after the stage has processed the data. For example, you might use a before-stage routine to prepare the data before processing starts. You might use an after-stage routine to send an electronic message when the stage has finished.

Editing Transformer Stages

The Transformer Editor enables you to perform the following operations on a Transformer stage:

- Create new columns on a link
- Delete columns from within a link
- Move columns within a link
- Edit column meta data
- Define output column derivations
- Define input column key expressions
- Specify before- and after-stage subroutines
- Define link constraints and handle rejects
- Specify the order in which links are processed
- Define local stage variables

Using drag-and-drop

Many of the Transformer stage edits can be made simpler by using the Transformer Editor's drag-and-drop functionality. You can drag columns from any link to any other link. Common uses are:

- Copying input columns to output links
- Moving columns within a link
- Copying derivations in output links
- Copying key expressions in input links.

To use drag-and-drop:

1. Click the source cell to select it.
2. Click the selected cell again and, without releasing the mouse button, drag the mouse pointer to the desired location within the target link. An insert point appears on the target link to indicate where the new cell will go.
3. Release the mouse button to drop the selected cell.

You can drag multiple columns, key expressions, or derivations. Use the standard Explorer keys when selecting the source column cells, then proceed as for a single cell.

You can drag and drop the full column set by dragging the link title.

You can add a column to the end of an existing derivation or key expression by holding down the **Ctrl** key as you drag the column.

Find and Replace Facilities

If you are working on a complex job where several links, each containing several columns, go in and out of the Transformer stage, you can use the find/replace column facility to help locate a particular column or expression and change it.

The find/replace facility enables you to:

- Find and replace a column name
- Find and replace expression text
- Find the next empty expression
- Find the next expression that contains an error

To use the find/replace facilities, open the Find and Replace dialog box by:

- Clicking the **Find/Replace** button on the toolbar
- Choosing **Find/Replace** from the link shortcut menu
- Pressing **Ctrl-F**

The Find and Replace dialog box has three tabs:

- **Expression Text**. Allows you to locate the occurrence of a particular string within an expression, and replace it if required. You can search up or down, and choose to match case, match whole words, or neither. You can also choose to replace all occurrences of the string within an expression.
- **Columns Names**. Allows you to find a particular column and rename it if required. You can search up or down, and choose to match case, match the whole word, or neither.
- **Expression Types**. Allows you to find the next empty expression or the next expression that contains an error. You can also press **Ctrl-M** to find the next empty expression or **Ctrl-N** to find the next erroneous expression.

  Note: The find and replace results are shown in the color specified in **Tools → Options**.

Press **F3** to repeat the last search you made without opening the **Find and Replace** dialog box.

Select Facilities

If you are working on a complex job where several links, each containing several columns, go in and out of the Transformer stage, you can use the select column facility to select multiple columns. This facility is also available in the **Mapping** tabs of certain parallel job stages.

The select facility enables you to:

- Select all columns/stage variables whose expressions contains text that matches the text specified.
- Select all column/stage variables whose name contains the text specified (and, optionally, matches a specified type).
- Select all columns/stage variable with a certain data type.
- Select all columns with missing or invalid expressions.

To use the select facilities, choose **Select** from the link shortcut menu. The Select dialog box appears. It has three tabs:

- **Expression Text**. This **Expression Text** tab allows you to select all columns/stage variables whose expressions contain text that matches the text specified. The text specified is a simple text match, taking into account the **Match case** setting.
- **Column Names**. The **Column Names** tab allows you to select all column/stage variables whose name contains the text specified. There is an additional **Data Type** drop down list, that will limit the columns selected to those with that data type. You can use the **Data Type** drop down list on its own to select all columns of a certain data type. For example, all string columns can be selected by leaving the text field blank, and selecting String as the data type. The data types in the list are generic data types, where each of the column SQL data types belong to one of these generic types.
- **Expression Types**. The **Expression Types** tab allows you to select all columns with either empty expressions or invalid expressions.

Specifying the Primary Input Link

The first link to a Transformer stage is always designated as the primary input link. However, you can choose an alternative link to be the primary link if necessary. To do this:

1. Select the current primary input link in the Diagram window.
2. Choose **Convert to Reference** from the Diagram window shortcut menu.
3. Select the reference link that you want to be the new primary input link.
4. Choose **Convert to Stream** from the Diagram window shortcut menu.

Creating and Deleting Columns

You can create columns on links to the Transformer stage using any of the following methods:

- Select the link, then click the **Load Column Definition** button in the toolbar to open the standard load columns dialog box.
- Use drag-and-drop or copy and paste functionality to create a new column by copying from an existing column on another link.
- Use the shortcut menus to create a new column definition.
- Edit the grids in the link's metadata tab to insert a new column.

When copying columns, a new column is created with the same metadata as the column it was copied from.

To delete a column from within the Transformer Editor, select the column you want to delete and click **Cut** or choose **Delete Column** from the shortcut menu.

Moving Columns Within a Link

You can move columns within a link using either drag-and-drop or cut and paste. Select the required column, then drag it to its new location, or cut it and paste it in its new location.

Editing Column Metadata

You can edit column metadata from within the grid in the bottom of the Transformer Editor. Select the tab for the link metadata that you want to edit, then use the standard IBM InfoSphere DataStage edit grid controls.

The metadata shown does not include column derivations or key expressions, since these are edited in the links area.

Defining Output Column Derivations

You can define the derivation of output columns from within the Transformer Editor in five ways:

- If you require a new output column to be directly derived from an input column, with no transformations performed, then you can drag or copy an input column to an output link. The output columns will have the same names as the input columns from which they were derived.
- If the output column already exists, you can drag or copy an input column to the output column's **Derivation** field. This specifies that the column is directly derived from an input column, with no transformations performed.
- You can use the column auto-match facility to automatically set that output columns are derived from their matching input columns.
- You might need one output link column derivation to be the same as another output link column derivation. In this case you can drag or copy the derivation cell from one column to another.
- In many cases you will need to transform data before deriving an output column from it. For these purposes you can use the Expression Editor. To display the Expression Editor, double-click on the required output link column **Derivation** cell. (You can also invoke the Expression Editor using the shortcut menu or the shortcut keys.)

If a derivation is displayed in red (or the color defined in **Tools → Options**), it means that the Transformer Editor considers it incorrect. (In some cases this might simply mean that the derivation does not meet the strict usage pattern rules of the server engine, but will actually function correctly.)

After an output link column has a derivation defined that contains any input link columns, a relationship line is drawn between the input column and the output column. There can be multiple relationship lines either in or out of columns. You can choose whether to view the relationships for all links, or just the relationships for the selected links, using the button in the toolbar.

Column Auto-Match Facility:

This time-saving feature allows you to automatically set columns on an output link to be derived from matching columns on an input link. Using this feature you can fill in all the output link derivations to route data from corresponding input columns, then go back and edit individual output link columns where you want a different derivation.

To use this facility:

1. Open the Column Auto-Match dialog box in one of the following ways:
   - Click the **Column Auto-Match** button in the Transformer Editor toolbar.
   - Choose **Auto Match** from the input link header or output link header shortcut menu.
2. Choose the input link and output link that you want to match columns for from the lists.
3. Click **Location match** or **Name match** from the **Match type** area.

If you choose **Location match**, this will set output column derivations to the input link columns in the equivalent positions. It starts with the first input link column going to the first output link column, and works its way down until there are no more input columns left.

If you choose **Name match**, you need to specify further information for the input and output columns as follows:

- **Input columns**:

  Match all columns or **Match selected columns**. Choose one of these to specify whether all input link columns should be matched, or only those currently selected on the input link.

  Ignore prefix. Optionally specifies characters at the front of the column name that should be ignored during the matching procedure.

  Ignore suffix. Optionally specifies characters at the end of the column name that should be ignored during the matching procedure.
- **Output columns**:

  Ignore prefix. Optionally specifies characters at the front of the column name that should be ignored during the matching procedure.

  Ignore suffix. Optionally specifies characters at the end of the column name that should be ignored during the matching procedure.
- **Ignore case**. Select this check box to specify that case should be ignored when matching names. The setting of this also affects the **Ignore prefix** and **Ignore suffix** settings. For example, if you specify that the prefix IP will be ignored, and turn **Ignore case** on, then both IP and ip will be ignored.

4. Click **OK** to proceed with the auto-matching.

   Note: Auto-matching does not take into account any data type incompatibility between matched columns; the derivations are set regardless.

Editing Multiple Derivations

You can make edits across several output column or stage variable derivations by choosing **Derivation Substitution...** from the shortcut menu. This opens the Expression Substitution dialog box.

The Expression Substitution dialog box allows you to make the same change to the expressions of all the currently selected columns within a link. For example, if you wanted to add a call to the trim() function around all the string output column expressions in a link, you could do this in two steps. First, use the Select dialog box to select all the string output columns. Then use the Expression Substitution dialog box to apply a trim() call around each of the existing expression values in those selected columns.

You are offered a choice between whole expression substitution and part of expression substitution.

Whole Expression:

With this option the whole existing expression for each column is replaced by the replacement value specified. This replacement value can be a completely new value, but will typically be a value based on the original expression value. When specifying the replacement value, the existing value of the column's expression can be included in this new value by including "$1". This can be included any number of times.

For example, when adding a trim() call around each expression of the currently selected column set, having selected the required columns, you would:

1. Select the **Whole expression** option.
2. Enter a replacement value of:

   ```
   trim($1)
   ```
3. Click **OK**

Where a column's original expression was:

```
DSLink3.col1
```

This will be replaced by:

```
trim(DSLink3.col1)
```

This is applied to the expressions in each of the selected columns.

If you need to include the actual text $1 in your expression, enter it as "$$1".

Part of Expression:

With this option, only part of each selected expression is replaced rather than the whole expression. The part of the expression to be replaced is specified by a Regular Expression match.

It is possible that more that one part of an expression string could match the Regular Expression specified. If **Replace all occurrences** is checked, then each occurrence of a match will be updated with the replacement value specified. If it is not checked, then just the first occurrence is replaced.

When replacing part of an expression, the replacement value specified can include that part of the original expression being replaced. In order to do this, the Regular Expression specified must have round brackets around its value. "$1" in the replacement value will then represent that matched text. If the Regular Expression is not surrounded by round brackets, then "$1" will simply be the text "$1".

For complex Regular Expression usage, subsets of the Regular Expression text can be included in round brackets rather than the whole text. In this case, the entire matched part of the original expression is still replaced, but "$1", "$2" and so on can be used to refer to each matched bracketed part of the Regular Expression specified.

Following is an example of the Part of expression replacement.

Suppose a selected set of columns have derivations that use input columns from `DSLink3'. For example, two of these derivations could be:

```
DSLink3.OrderCount + 1
If (DSLink3.Total > 0) Then DSLink3.Total Else -1
```

You might want to protect the usage of these input columns from null values, and use a zero value instead of the null. To do this:

1. Select the columns you want to substitute expressions for.
2. Select the **Part of expression** option.
3. Specify a Regular Expression value of:

   ```
   (DSLink3\.[a-z,A-Z,0-9]*)
   ```

   This will match strings that contain "DSLink3.", followed by any number of alphabetic characters or digits. (This assumes that column names in this case are made up of alphabetic characters and digits). The round brackets around the whole Expression means that $1 will represent the whole matched text in the replacement value.
4. Specify a replacement value of

   ```
   NullToZero($1)
   ```

   This replaces just the matched substrings in the original expression with those same substrings, but surrounded by the NullToZero call.
5. Click **OK**, to apply this to all the selected column derivations.

From the examples above:

```
DSLink3.OrderCount + 1
```

would become

```
NullToZero(DSLink3.OrderCount) + 1
```

and

```
If (DSLink3.Total > 0) Then DSLink3.Total Else -1
```

would become:

```
If (NullToZero(DSLink3.Total) > 0) Then DSLink3.Total Else -1
```

If the **Replace all occurrences** option is selected, the second expression will become:

```
If (NullToZero(DSLink3.Total) > 0)
Then NullToZero(DSLink3.Total)
Else -1
```

The replacement value can be any form of expression string. For example in the case above, the replacement value could have been:

```
(If (StageVar1 > 50000) Then $1 Else ($1 + 100))
```

In the first case above, the expression

```
DSLink3.OrderCount + 1
```

would become:

```
(If (StageVar1 > 50000) Then DSLink3.OrderCount
Else (DSLink3.OrderCount + 100)) + 1
```

Defining Input Column Key Expressions

You can define key expressions for key fields of reference inputs. This is similar to defining derivations for output columns.

In most cases a key expression will be an equijoin from a primary input link column. You can specify an equijoin in two ways:

- Use drag-and-drop to drag a primary input link column to the appropriate key expression cell.
- Use copy and paste to copy a primary input link column and paste it on the appropriate key expression cell.

A relationship link is drawn between the primary input link column and the key expression.

You can also drag or copy an existing key expression to another input column, and you can drag or copy multiple selections.

If you require a more complex expression than an equijoin, then you can double-click the required key expression cell to open the Expression Editor.

If a key expression is displayed in red (or the color defined in **Tools → Options**), it means that the Transformer Editor considers it incorrect. (In some cases this might simply mean that the key expression does not meet the strict usage pattern rules of the server engine, but will actually function correctly.)

Initially, key expression cells occupy a very narrow column. In most cases the relationship line gives sufficient information about the key expression, but otherwise you can drag the left edge of the column to expand it.

Defining Multirow Lookup for Reference Inputs

Where a reference link originates from a UniVerse or ODBC stage, you can look up multiple rows from the reference table. The rows are selected by a foreign key rather than a primary key, as is the case for normal reference links.

In order to use the multirow functionality, you must define which column or columns are the foreign keys in the column metadata. Do this by changing the **Key** attribute for the current primary key column to **No** and then change the **Key** attribute for the required foreign key column, or columns, to **Yes**. The foreign key expressions can then be defined through the Expression Editor, as with normal primary key expressions described in "Defining Input Column Key Expressions" on page 108.

You also need to specify that the reference link uses the multirow functionality.

Do this by opening the **Transformer Stage Properties** dialog box, go to the **General** tab on the Inputs page (making sure the reference input link is selected) and select the **reference link with multi row result set** check box.

Specifying Before-Stage and After-Stage Subroutines

Because the Transformer stage is an active stage type, you can specify routines to be executed before or after the stage has processed the data.

To specify a routine, click the stage properties button in the toolbar to open the **Stage Properties** dialog box. The **General** tab contains the following fields:

- **Before-stage subroutine** and **Input Value**. Contain the name (and value) of a subroutine that is executed before the stage starts to process any data.
- **After-stage subroutine** and **Input Value**. Contain the name (and value) of a subroutine that is executed after the stage has processed the data.

Choose a routine from the list. This list contains all the built routines defined as a **Before/After Subroutine** in the **Routines** folder in the repository tree. Enter an appropriate value for the routine's input argument in the **Input Value** field.

If you choose a routine that is defined in the repository, but which was edited but not compiled, a warning message reminds you to compile the routine when you close the **Transformer stage** dialog box.

If you installed or imported a job, the **Before-stage subroutine** or **After-stage subroutine** field might reference a routine that does not exist on your system. In this case, a warning message appears when you close the dialog box. You must install or import the "missing" routine or choose an alternative one to use.

A return code of 0 from the routine indicates success, any other code indicates failure and causes a fatal error when the job is run.

If you edit a job created using Release 1 of IBM InfoSphere DataStage, the **Before-stage subroutine** or **After-stage subroutine** field might contain the name of a routine created at Release 1. When InfoSphere DataStage is upgraded, these routines are identified and automatically renamed. For example, if you used a before-stage subroutine called BeforeSubr, this appears as BeforeSubr\<Rev1> in the **Before-stage subroutine** field. You can continue to use these routines. However, because you could not specify input values for routines at Release 1 of InfoSphere DataStage, the **Input Value** field grays out when you use one of these "old" routines.

Defining Constraints and Handling Rejects

You can define limits for output data by specifying a constraint. Constraints are BASIC expressions and you can specify a constraint for each output link from a Transformer stage. You can also specify that a particular link is to act as a reject link. Reject links output rows that have not been written on any other output links from the Transformer stage.

To define a constraint or specify a reject link, use one of the following options:

- Select an output link and click the **Constraints** button.
- Double-click the output link's constraint entry field.
- Choose **Constraints** from the background or header shortcut menus.

A dialog box appears which allows you either to define constraints for any of the Transformer output links or to define a link as a reject link.

Define a constraint by entering a BASIC expression in the **Constraint** field for that link. After you have done this, any constraints will appear below the link's title bar in the Transformer Editor. This constraint expression will then be checked against the row data at runtime. If the data does not satisfy the constraint, the row will not be written to that link. It is also possible to define a link which can be used to catch these rows which have been "rejected" from a previous link.

A reject link can be defined by choosing **Yes** in the **Reject Row** field and setting the **Constraint** field as follows:

- To catch rows which are rejected from a specific output link, set the **Constraint** field to *linkname*.REJECTED. This will be set whenever a row is rejected on the *linkname* link, whether because the row fails to match a constraint on that output link, or because a write operation on the target fails for that row. Note that such a reject link should occur *after* the output link from which it is defined to catch rejects.
- To catch rows which caused a write failure on an output link, set the **Constraint** field to *linkname*.REJECTEDCODE. The value of *linkname*.REJECTEDCODE will be non-zero if the row was rejected due to a write failure or 0 (DSE.NOERROR) if the row was rejected due to the link constraint not being met. When editing the **Constraint** field, you can set return values for *linkname*.REJECTEDCODE by selecting from the Expression Editor **Link Variables►Constants...** menu options. These give a range of errors, but note that most write errors return DSE.WRITERROR.

  In order to set a reject constraint which differentiates between a write failure and a constraint not being met, a combination of the *linkname*.REJECTEDCODE and *linkname*.REJECTED flags can be used. For example:

  – To catch rows which have failed to be written to an output link, set the **Constraint** field to *linkname*.REJECTEDCODE
  – To catch rows which do not meet a constraint on an output link, set the **Constraint** field to *linkname*.REJECTEDCODE = DSE.NOERROR AND *linkname*.REJECTED
  – To catch rows which have been rejected due to a constraint or write error, set the **Constraint** field to *linkname*.REJECTED
- As a "catch all," the **Constraint** field can be left blank. This indicates that this reject link will catch all rows which have not been successfully written to *any* of the output links processed up to this point. Therefore, the reject link should be the last link in the defined processing order.
- Any other **Constraint** can be defined. This will result in the number of rows written to that link (that is, rows which satisfy the constraint) to be recorded in the job log as "rejected rows."

  Note: Due to the nature of the "catch all" case above, you should only use one reject link whose **Constraint** field is blank. To use multiple reject links, you should define them to use the *linkname*.REJECTED flag detailed in the first case above.

Specifying Link Order

You can specify the order in which both input and output links process a row. For input links, you can order reference links (the primary link is always processed first). For output links, you can order all the links.

The initial order of the links is the order in which they are added to the stage.

To reorder the links:

1. Open the **Link Ordering** tab of the **Transformer Stage Properties** dialog box in one of these ways:
   - Click the **Input Link Execution Order** or **Output Link Execution Order** button on the Transformer Editor toolbar.
   - Choose **Reorder input links** or **Reorder output links** from the background shortcut menu.
   - Click the **Stage Properties** button in the Transformer toolbar or choose **Stage Properties** from the background shortcut menu and click on the Stage page **Link Ordering** tab.
2. Use the arrow buttons to rearrange the list of links in the execution order required.
3. When you are happy with the order, click **OK**.

   Note: Although the link ordering facilities mean that you can use a previous output column to derive a subsequent output column, this is not advised and you will receive a warning if you do so.

Defining Local Stage Variables

You can declare and use your own variables within a Transformer stage. Such variables are accessible only from the Transformer stage in which they are declared. They can be used as follows:

- They can be assigned values by expressions.
- They can be used in expressions which define an output column derivation.
- Expressions evaluating a variable can include other variables or the variable being evaluated itself.

Any stage variables you declare are shown in a table in the right pane of the links area. The table looks similar to an output link. You can display or hide the table by clicking the **Stage Variables** button in the Transformer toolbar or choosing **Stage Variables** from the background shortcut menu. Stage variables are not shown in the output link metadata area at the bottom of the right pane.

The table lists the stage variables together with the expressions used to derive their values. Link lines join the stage variables with input columns used in the expressions. Links from the right side of the table link the variables to the output columns that use them.

To declare a stage variable:

1. Open the Transformer Stage Properties dialog box in either of these ways:
   - Click the **Stage Properties** button in the Transformer toolbar.
   - Choose **Stage Properties** from the background shortcut menu.
2. Click the **Variables** tab on the General page. The **Variables** tab contains a grid showing currently declared variables, their initial values, and an optional description. Use the standard grid controls to add new variables. Variable names must begin with an alphabetic character (a-z, A-Z) and can only contain alphanumeric characters (a-z, A-Z, 0-9). Ensure that the variable does not use the name of any BASIC keywords.

Variables entered in the **Stage Properties** dialog box appear in the Stage Variable table in the links pane.

You perform most of the same operations on a stage variable as you can on an output column (see "Defining Output Column Derivations" on page 105). A shortcut menu offers the same commands. You cannot, however, paste a stage variable as a new column or a column as a new stage variable.

The IBM InfoSphere DataStage Expression Editor

The InfoSphere DataStage Expression Editor helps you to enter correct expressions when you edit Transformer stages. It also helps you to define custom transforms in the repository (see "Defining Custom Transforms" on page 133). The Expression Editor can:

- Facilitate the entry of expression elements
- Complete the names of frequently used variables
- Validate variable names and the complete expression

The Expression Editor can be opened from:

- Output link **Derivation** cells
- Stage variable **Derivation** cells
- Input link **Key Expression** cells
- Constraint dialog box
- Transform dialog box in the repository

Expression Format

The format of an expression is as follows:

```
KEY:

something_like_this       is a token
something_in_italics      is a terminal, that is, doesn't break down any
   further
|     is a choice between tokens
[     is an optional part of the construction
"XXX"     is a literal token (that is, use XXX not
   including the quotation marks)
================================================
expression ::=    function_call |
      variable_name |
   other_name |
   constant |
   unary_expression |
   binary_expression |
   if_then_else_expression |
   substring_expression |
   "(" expression ")"

function_call ::= function_name "(" [argument_list] ")"
argument_list ::= expression | expression "," argument_list
function_name ::=    name of a built-in function |
   name of a user-defined function
variable_name ::=       job_parameter name |
      stage_variable_name |
      link_variable name
other_name ::=    name of a built-in macro, system variable, and so on
constant ::= numeric_constant | string_constant
numeric_constant ::= ["+" | "-"] digits ["." [digits]] ["E" | "e" ["+" | "-"] digits]
string_constant ::=    "'" [characters] "'" |
   """ [characters] """ |
   "\" [characters] "\"
unary_expression ::= unary_operator expression
unary_operator ::= "+" | "-"
binary_expression ::= expression binary_operator expression
binary_operator ::=    arithmetic_operator |
   concatenation_operator |
   matches_operator |
   relational_operator |
   logical_operator
arithmetic_operator ::= "+" | "-" | "*" | "/" | "^"
concatenation_operator ::= ":"
```

```
matches_operator ::= "MATCHES"
relational_operator ::=    "   =" |"EQ" |
      "<>" | "#" | "NE" |
      ">" | "GT" |
      ">=" | "=>" | "GE" |
      "<" | "LT" |
      "<=" | "=<" | "LE"
logical_operator ::= "AND" | "OR"
if_then_else_expression ::= "IF" expression "THEN" expression "ELSE" expression
substring_expression ::= expression "[" [expression ["," expression] "]"
field_expression ::=    expression "[" expression ","
   expression ","
   expression "]"
         /*    That is, always 3 args
```

Note: Keywords like "AND" or "IF" or "EQ" can be in any case.

Entering Expressions

Whenever the insertion point is in an expression box, you can use the Expression Editor to suggest the next element in your expression. Do this by right-clicking in the box, or by clicking the **Suggest** button to the right of the box. This opens the **Suggest Operand** or **Suggest Operator** menu. Which menu appears depends on context, that is, whether you should be entering an operand or an operator as the next expression element.

You will be offered a different selection on the **Suggest Operand** menu depending on whether you are defining key expressions, derivations and constraints, or a custom transform. The **Suggest Operator** menu is always the same.

Suggest Operand Menu - Transformer Stage
DS Macro...
DS Function...
DS Constant...
DS Routine...
DS Transform...
Job Parameter...
Input Column...
Link Variables
Stage Variables...
System Variable...
String...
Function...
() Parentheses
If Then Else

Suggest Operand Menu - Defining Custom Transforms
DS Macro...
DS Function...
DS Constant...
DS Routine...
Transform Argument...

Suggest Operand Menu - Defining Custom Transforms
System Variable...
String...
Function...
() Parentheses
If Then Else

Suggest Operator Menu		
+	=	Concatenate
-	<>	Substring
*	<	Matches
/	<=	And
^	>	Or
	>=	

Completing Variable Names

The Expression Editor stores variable names. When you enter a variable name you have used before, you can type the first few characters, then press **F5**. The Expression Editor completes the variable name for you.

If you enter the name of an input link followed by a period, for example, `DailySales.`, the Expression Editor displays a list of the column names of that link. If you continue typing, the list selection changes to match what you type. You can also select a column name using the mouse. Enter a selected column name into the expression by pressing **Tab** or **Enter**. Press **Esc** to dismiss the list without selecting a column name.

Validating the Expression

When you have entered an expression in the Transformer Editor, press **Enter** to validate it. The Expression Editor checks that the syntax is correct and that any variable names used are acceptable to the compiler. When using the Expression Editor to define a custom transform, click **OK** to validate the expression.

If there is an error, a message appears and the element causing the error is highlighted in the expression box. You can either correct the expression or close the Transformer Editor or Transform dialog box.

Within the Transformer Editor, the invalid expressions are shown in red. (In some cases this might simply mean that the expression does not meet the strict usage pattern rules of the server engine, but will actually function correctly.)

For more information about the syntax you can use in an expression, see Chapter 7, “BASIC Programming,” on page 137.

Exiting the Expression Editor

You can exit the Expression Editor in the following ways:

- Press **Esc** (which discards changes).
- Press **Return** (which accepts changes).
- Click outside the Expression Editor box (which accepts changes).

Configuring the Expression Editor

The Expression Editor is switched on by default. If you prefer not to use it, you can switch it off or use selected features only. The Expression Editor is configured by editing the Designer client options. For more information about Designer client options, see *IBM InfoSphere DataStage and QualityStage Designer Client Guide*.

Transformer Stage Properties

The Transformer stage has a Properties dialog box which allows you to specify details about how the stage operates.

The Transformer Stage dialog box has three pages:

- Stage page. This is used to specify general information about the stage.
- **Inputs** page. This is where you specify details about the data input to the Transformer stage.
- **Outputs** page. This is where you specify details about the output links from the Transformer stage.

Stage Page

The Stage page has four tabs:

- **General**. Allows you to enter an optional description of the stage and specify a before-stage or after-stage subroutine.
- **Variables**. Allows you to set up stage variables for use in the stage.
- **Link Ordering**. Allows you to specify the order in which the output links will be processed.

The **General** tab is described in "Before-Stage and After-Stage Routines" on page 102. The **Variables** tab is described in "Defining Local Stage Variables" on page 111. The **Link Ordering** tab is described in "Specifying Link Order" on page 111.

Inputs Page

The Inputs page allows you to specify details about data coming into the Transformer stage. The Transformer stage can have only one input link.

The **General** tab allows you to specify an optional description of the input link.

Outputs Page

The Outputs page has a **General** tab which allows you to enter an optional description for each of the output links on the **Transformer** stage.

Chapter 5. Debugging and Compiling a Job

These topics describe how to create an executable job. When you have edited all the stages in a job design, you can create an executable job by compiling your job design. The debugger helps you to iron out any problems in your design. The job can then be validated and run using the Director client.

The IBM InfoSphere DataStage Debugger

The InfoSphere DataStage debugger provides basic facilities for testing and debugging your job designs. The debugger is run from the Designer client. It can be used from a number of places within the Designer client:

- **Debug** menu (**Debug**)
- Debug toolbar
- Shortcut menu (some commands).

The debugger enables you to set breakpoints on the links in your job. When you run the job in debug mode, the job will stop when it reaches a breakpoint. You can then step to the next action (reading or writing) on that link, or step to the processing of the next row of data (which might be on the same link or another link).

Any breakpoints you have set remain if the job is closed and reopened. Breakpoints are validated when the job is compiled, and remain valid if the link to which it belongs is moved, or has either end moved, or is renamed. If, however, a link is deleted and another of the same name created, the new link does not inherit the breakpoint. Breakpoints are not inherited when a job is saved under a different name, exported, or upgraded.

Note: You should be careful when debugging jobs that do parallel processing (using IPC stages or interprocess active-to-active links). You cannot set breakpoints on more than one process at a time. To ensure this doesn't happen, you should only set one breakpoint at a time in such jobs.

To add a breakpoint:

1. Select the required link.
2. Choose **Toggle Breakpoint** from the **Debug** menu or the Debug toolbar. The breakpoint can subsequently be removed by choosing **Toggle Breakpoint** again.

A circle appears on the link to indicate that a breakpoint has been added. Choose **Edit Breakpoints** from the **Debug** menu, or click the **Edit Breakpoints** button in the Debug toolbar to open the Edit Breakpoints dialog box and set up the breakpoint.

You cannot place a breakpoint on a link which has a container as its source stage. Instead, you should place the breakpoint on the same link as represented within the container view itself. The link will only be shown as having a breakpoint in the container view. For more information see “Debugging Shared Containers” on page 118.

The Debug Window allows you to view variables in the watch list and any in-context variables when you stop at a breakpoint.

The Debug Window is visible whenever **Debug → Debug Window** is selected. It always appears on the top of the Designer client window. Right-clicking in the Debug Window displays a shortcut menu containing the same items as the **Debug** menu. The Debug Window has two display panes. You can drag

the splitter bar between the two panes to resize them relative to one another. The window also gives information about the status of the job and debugger.

The upper pane shows local variables. Before debugging starts, all the columns on all the links in the job are displayed, and all are marked "Out of context." During debugging, the pane shows only the variables that are in context when the job is stopped at a breakpoint. It displays the names and values of any variables currently in context and you can add any of these variables to the watch list, which maintains a record of selected variables for as long as required.

The lower pane displays variables in the watch list. When variables are in context, their values are displayed and updated at every breakpoint. When variables are out of context, they are marked "Out of context." The watch list is saved between sessions.

To add a variable to the watch list:

1. Select the variable name in the upper pane of the Debug Window.
2. Click **Add Watch**. The variable will be added to the watch list and will appear in the lower pane.

To delete variables from the watch list, select the variables and click Remove Watch.

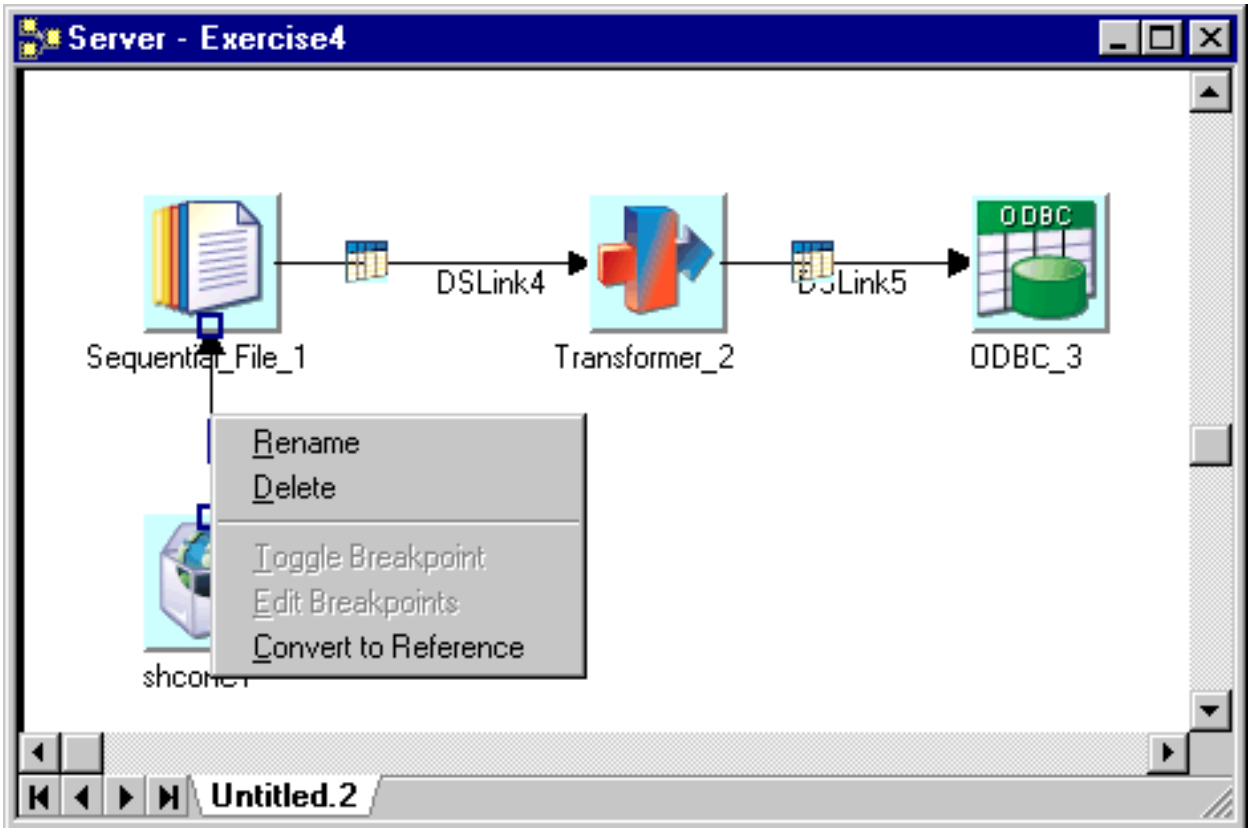
The following commands are available from the **Debug** menu or Debug toolbar:

- **Target Job**. Selects the job to debug. Only one job can be debugged at any one time.
  After a job has been debugged, the job in the **Target Job** list will not be available.
- **Go**. Runs the current job in debug mode, compiling it first if necessary. In debug mode the job will run until a breakpoint is encountered. It then stops in break mode, allowing you to interact with the job. The first time that **Go** is used after a job is compiled or loaded, the Job Run Options dialog box appears and collects any required parameter values or runtime limits.
- **Step to Next Link**. This causes the job to run until the next action occurs on any link (reading or writing), when it stops in break mode.
- **Step to Next Row**. This causes the job to run until the next row is processed or until another link with a breakpoint is encountered, whichever comes first. The job then stops in break mode. If the job is not currently stopped at a breakpoint on a link (for example, if it hasn't started debugging yet, or is stopped at a warning), then this will perform as **Step to Next Link**.
- **Stop Job**. Only available in break mode. Stops the job and exits break mode.
- **Job Parameters...** . Allows you to specify job parameters for when the job is run in debug mode. Selecting this invokes the **Job Run Options** dialog box, allowing you to specify any required parameters or runtime limits for the job. The item is disabled when the job is started in debug mode.
- **Edit Breakpoints...** . Allows you to edit existing breakpoints or add new ones.
- **Toggle Breakpoint**. Allows you to set or clear a breakpoint from the selected link. If a link has a breakpoint set (indicated by a dark circle at the link source), then **Toggle Breakpoint** clears that breakpoint. If the link has no breakpoint, then one is added, specifying a stop at every row processed.
- **Clear All Breakpoints**. Deletes all breakpoints defined for all links.
- **View Job Log**. Select this to open the Director client with the current job open in the job log view (the job must have been saved in the Designer client at some point for this to work)
- **Debug Window**. Select this to display the Debug Window. Clear it to hide the Debug Window.

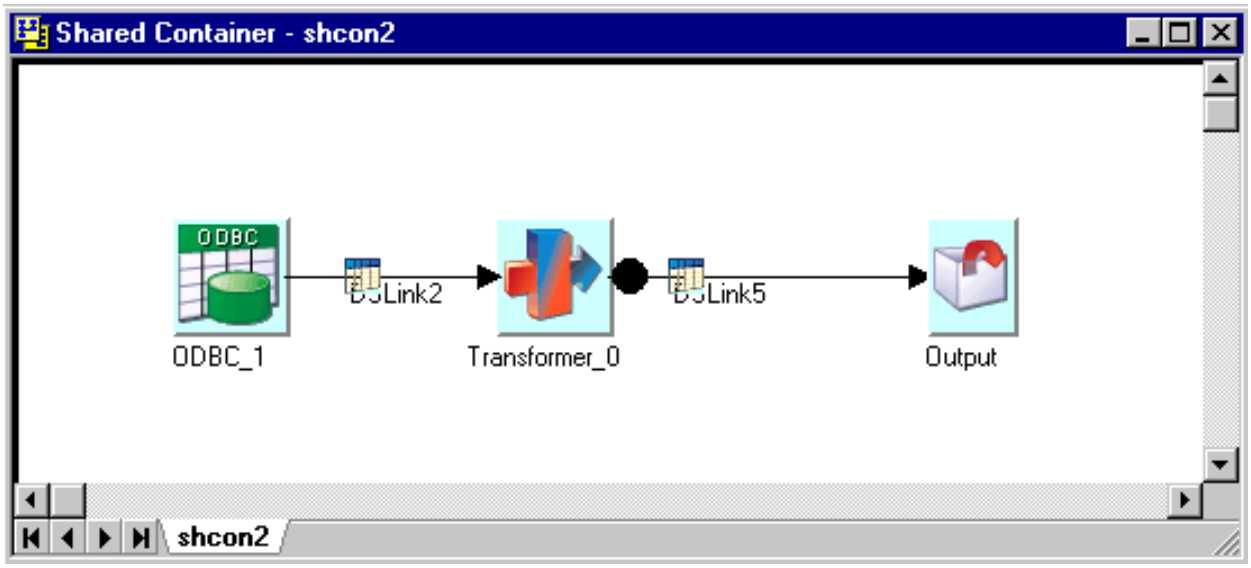
Debugging Shared Containers

The process for debugging Shared Containers is the same as that for other jobs, but breakpoints are handled differently:

- You cannot place a breakpoint on a link which has a container as its source stage.



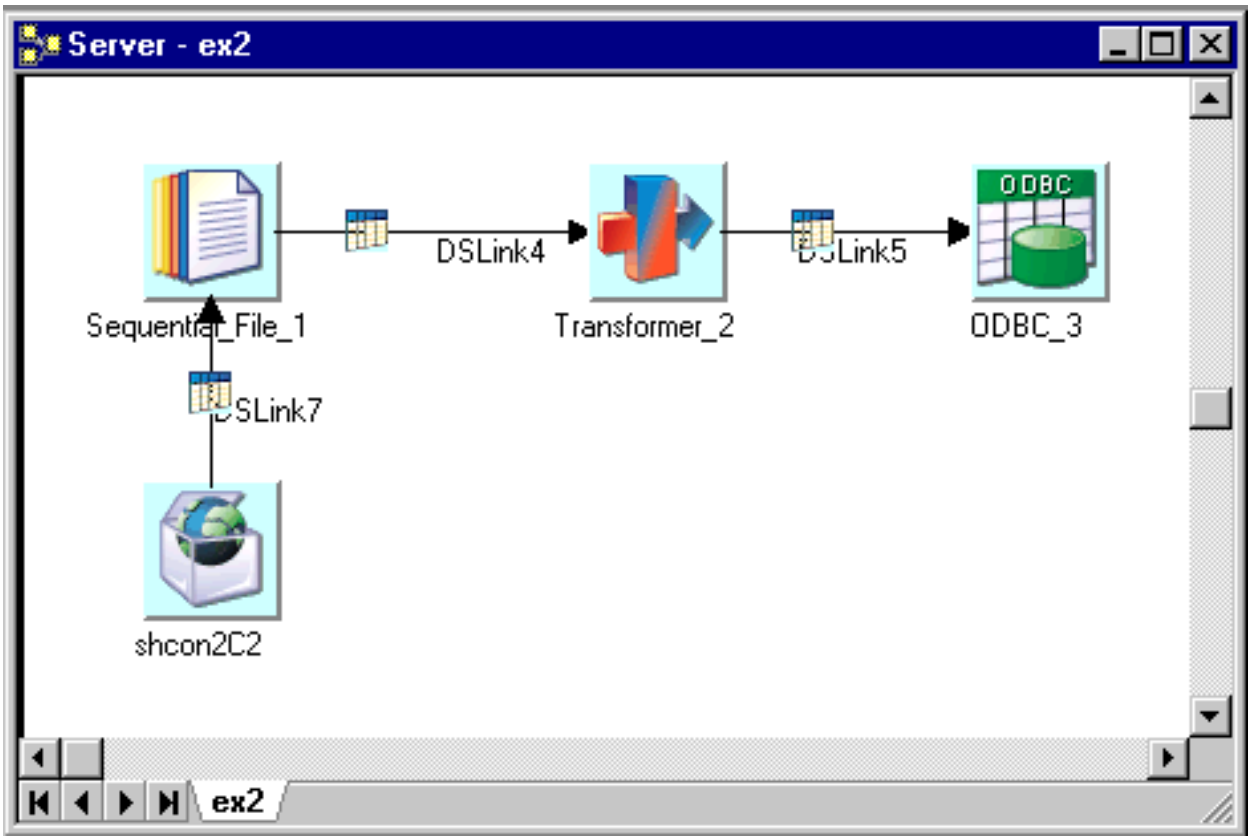
Instead, you should place the breakpoint on the same link as represented within the container view. The link will only be shown as having a breakpoint in the container view.



- If a breakpoint is set on a link inside a Shared Container, it will only become active (and visible) for the target job as shown on the debug bar.

Note: The debug bar only shows open Server Jobs because a Shared Container cannot be run outside the context of a job.

- If a different job uses the same shared container that is being debugged, then the breakpoint will not be visible or be hit in the other job. The example below shows a job called `Ex2' which uses the same shared container as the previous example called `Exercise 4.' The breakpoint will only be set for the target job which is Exercise 4.



Compiling a Job

Jobs are compiled using the Designer client. To compile a job, open the job in the Designer client and do one of the following:

- Choose **File** → **Compile**.
- Click the **Compile** button on the toolbar.

If the job has unsaved changes, you are prompted to save the job by clicking **OK**. The Compile Job window opens. This window contains a display area for compilation messages and has the following buttons:

- **Re-Compile**. Recompiles the job if you have made any changes.
- **Show Error**. Highlights the stage that generated a compilation error. This button is only active if an error is generated during compilation.
- **More**. Displays the output that does not fit in the display area. Some errors produced by the compiler include detailed BASIC output.
- **Close**. Closes the Compile Job window.
- **Help**. Invokes the help system.

The job is compiled as soon as this window opens. You must check the display area for any compilation messages or errors that are generated.

If breakpoints are set for links that no longer exist, a message appears during compilation to warn you about this. The breakpoints are then automatically removed.

You can also compile multiple jobs at once using the IBM InfoSphere DataStage compiler wizard. See *IBM InfoSphere DataStage and QualityStage Designer Client Guide* for more information.

Compilation Checks

During compilation, the following criteria in the job design are checked:

- **Primary Input**. If you have more than one input link to a Transformer stage, the compiler checks that one is defined as the primary input link.
- **Reference Input**. If you have reference inputs defined in a Transformer stage, the compiler checks that these are not from sequential files.
- **Key Expressions**. If you have key fields specified in your column definitions, the compiler checks that there are key expressions joining the data tables.
- **Transforms**. If you have specified a transform, the compiler checks that this is a suitable transform for the data type.

Successful Compilation

If the Compile Job window displays the message `Job successfully compiled with no errors` you can:

- Validate the job
- Run or schedule the job

Jobs are validated and run using the Director client. (You can also test run a job in the Designer client during development, but production runs are typically performed in the Director client.) See *IBM InfoSphere DataStage and QualityStage Director Client Guide* for more information.

Troubleshooting

If the Compile Job window displays an error, you can use the **Show Error** button to troubleshoot your job design. When you click **Show Error**, the stage that contains the first error in the design is highlighted. You must edit the stage to change any incorrect settings and recompile.

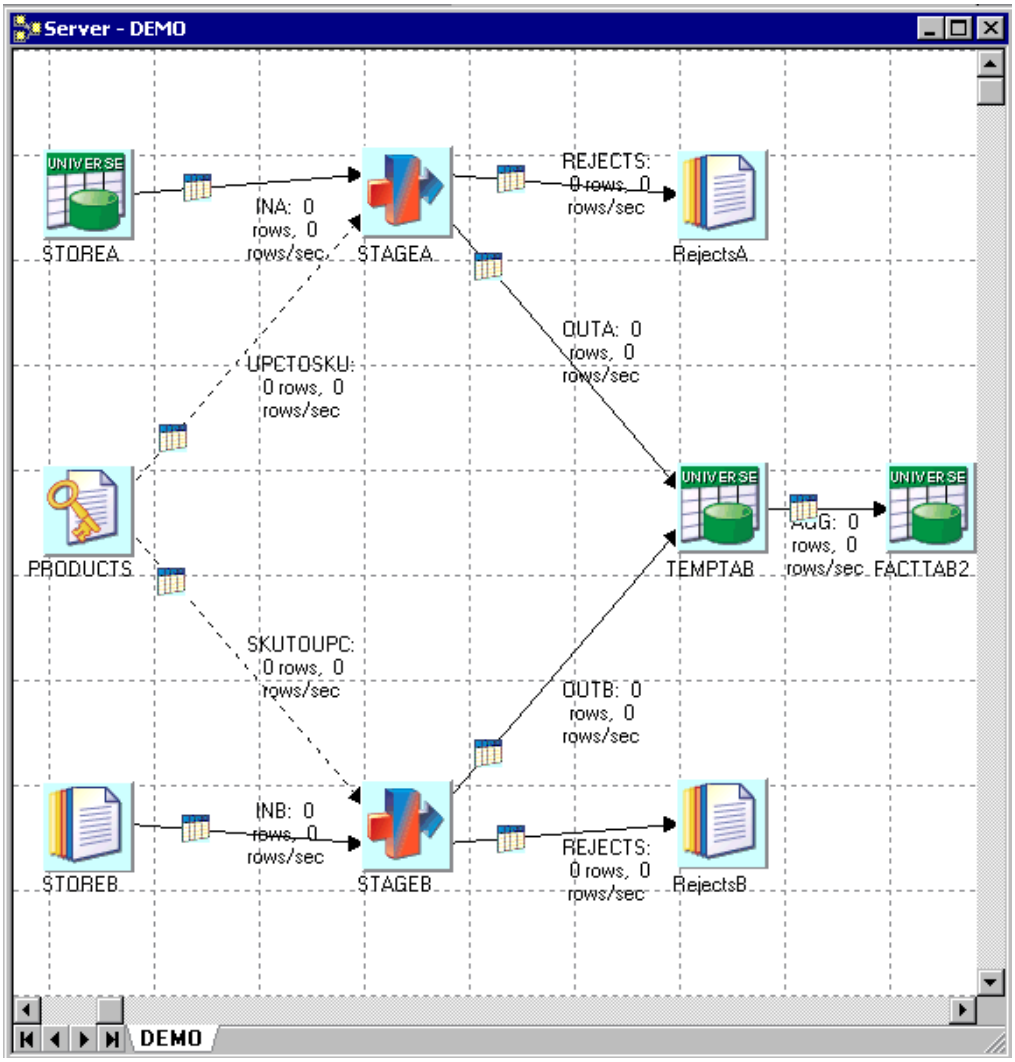
The process of troubleshooting compilation errors is an iterative process. You must refine each "problem" stage until the job compiles successfully.

Graphical Performance Monitor

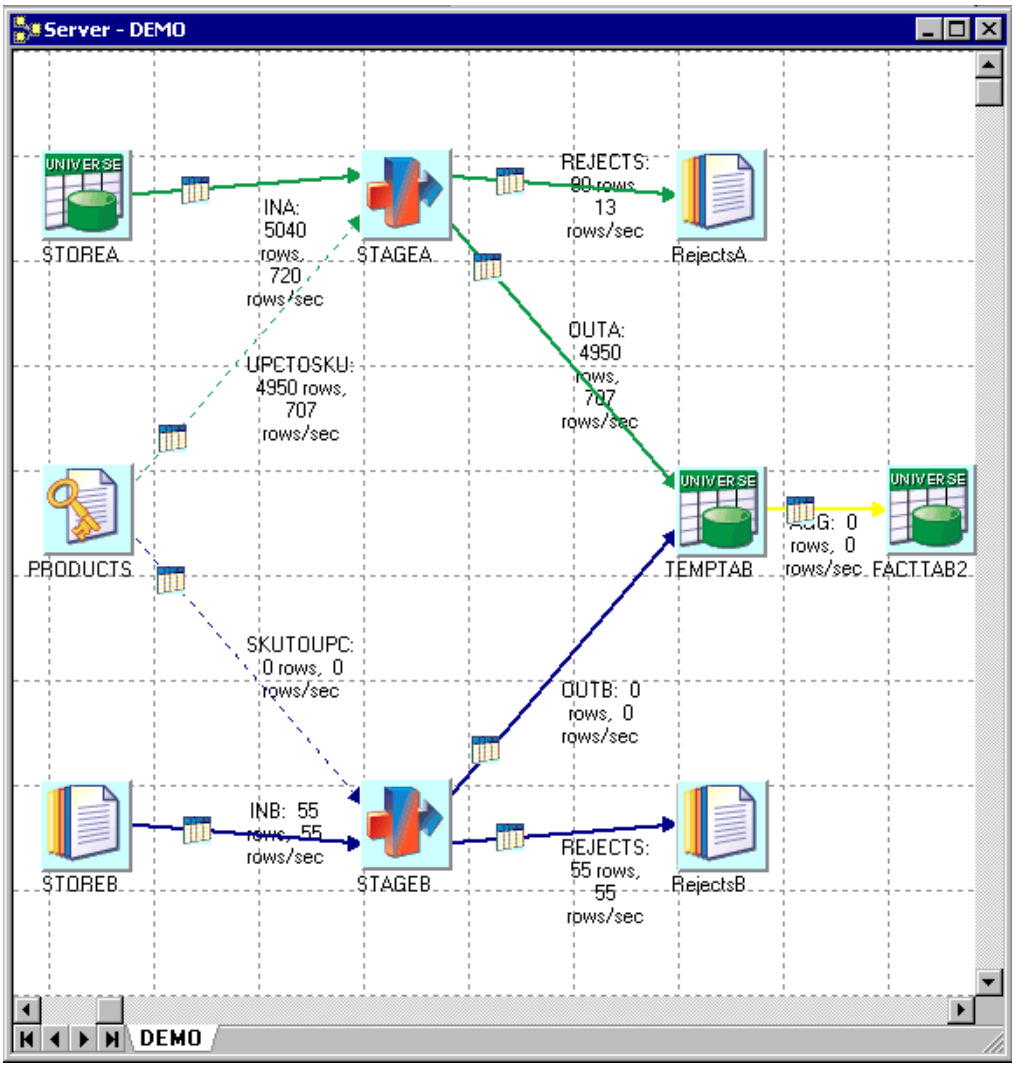
The performance monitor is a useful diagnostic aid when you design IBM InfoSphere DataStage server jobs. When you turn it on and compile a job, it displays information against each link in the job. When you run the job, either through the Director client or the debugger, the link information is populated with statistics to show the number of rows processed on the link and the speed at which they were processed. The links change color as the job runs to show the progress of the job.

To use the performance monitor:

1. With the job open and compiled in the Designer client, choose **Diagram → Show performance statistics**. Performance information appears against the links. If the job has not yet been run, the figures will be empty.



2. Run the job (either from the Director client or by choosing **Debug** → **Go**). Watch the links change color as the job runs and the statistics are populated with number of rows and rows/sec.



If you alter anything on the job design, you will lose the statistical information until the next time you compile the job.

The colors that the performance monitor uses are set via the Options dialog box. Choose **Tools → Options** and select the **Graphical Performance Monitor** branch to view the default colors and change them if required.

You can also set the refresh interval at which the monitor updates the information while the job is running.

Chapter 6. Programming in IBM InfoSphere DataStage

These topics describe the programming tasks that you can perform in InfoSphere DataStage server jobs. Most of these use the BASIC language, which provides you with a powerful procedural programming tool.

There are several areas within a server job where you might want to enter some code:

- Defining custom routines to use as building blocks within other programming tasks. For example, you can define a routine which will then be reused by several custom transforms. You can view, edit, and create your own BASIC routines using the Designer client.
- Defining custom transforms. The function specified in a transform definition converts the data in a chosen column.
- Defining derivations, key expressions, and constraints while editing a Transformer stage.
- Defining before-stage and after-stage subroutines. These subroutines perform an action before or after a stage has processed data. These subroutines can be specified for Aggregator, Transformer, and some supplemental stages.
- Defining before-job and after-job subroutines. These subroutines perform an action before or after a job is run and are set as job properties.
- Defining job control routines. These subroutines can be used to control other jobs from within the current job.

Programming Components

There are different types of programming components used within server jobs. They fall within three broad categories:

- **Built-in**. IBM InfoSphere DataStage comes with several built-in programming components that you can reuse within your server jobs as required. Some of the built-in components are accessible from the repository, and you can copy code from these. Others are only accessible from the Expression Editor, and the underlying code is not visible.
- **Custom**. You can also define your own programming components using the Designer client, specifically routines (see "Working with Routines" on page 127) and custom transforms (see "Defining Custom Transforms" on page 133). These are stored in the repository and can be reused for other jobs and by other InfoSphere DataStage users.
- **External**. You can use certain types of external component from within InfoSphere DataStage. If you have a large investment in custom UniVerse functions or ActiveX (OLE) functions, then it is possible to call these from within InfoSphere DataStage. This is done by defining a wrapper routine which in turn calls the external functions. Note that the mechanism for including custom UniVerse functions is different from including ActiveX (OLE) functions.

The following sections discuss programming terms you will come across when programming server jobs.

Routines

Routines are stored in the **Routines** folder in the repository tree by default, but you can store them in any folder you choose. You create, view or edit routines using the **Server** Routine dialog box. The following program components are classified as routines:

- **Transform functions**. These are functions that you can use when defining custom transforms. IBM InfoSphere DataStage has a number of built-in transform functions which are located in the **Routines** → **Examples** → **Functions** folder in the repository tree. You can also define your own transform functions in the **Server** Routine dialog box.

- **Before/After subroutines**. When designing a job, you can specify a subroutine to run before or after the job, or before or after an active stage. InfoSphere DataStage has a number of built-in before/after subroutines, which are located in the **Routines►Built-in → Before/After** folder in the repository tree. You can also define your own before/after subroutines using the **Server** Routine dialog box.
- **Custom UniVerse functions**. These are specialized BASIC functions that have been defined outside InfoSphere DataStage. Using the **Server** Routine dialog box, you can get InfoSphere DataStage to create a wrapper that enables you to call these functions from within InfoSphere DataStage. These functions are stored in the **Routines** folder in the repository tree. You specify the category when you create the routine. If NLS is enabled, you should be aware of any mapping requirements when using custom UniVerse functions. If a function uses data in a particular character set, it is your responsibility to map the data to and from Unicode.
- **ActiveX (OLE) functions**. You can use ActiveX (OLE) functions as programming components within InfoSphere DataStage. Such functions are made accessible to InfoSphere DataStage by importing them. This creates a wrapper that enables you to call the functions. After import, you can view and edit the BASIC wrapper using the **Server** Routine dialog box. By default, such functions are located in the **Routines → Class name** folder in the repository tree, but you can specify your own folder when importing the functions.

When using the Expression Editor, all of these components appear under the DS Routines... command on the **Suggest Operand** menu.

A special case of routine is the job control routine. Such a routine is used to set up a job that controls other jobs. Job control routines are specified in the Job control page in the Job Properties dialog box. Job control routines are not stored in the **Routines** folder in the repository tree.

Transforms

Transforms are stored in the **Transforms** folder in the repository tree by default, but you can store them in any folder you choose. You create, view or edit transforms using the **Transform** dialog box. Transforms specify the type of data transformed, the type it is transformed into and the expression that performs the transformation.

IBM InfoSphere DataStage is supplied with a number of built-in transforms (which you cannot edit). You can also define your own custom transforms, which are stored in the repository and can be used by other jobs.

When using the Expression Editor, the transforms appear under the DS Transform... command on the **Suggest Operand** menu.

Functions

Functions take arguments and return a value. The word "function" is applied to many components in IBM InfoSphere DataStage:

- **BASIC functions**. These are one of the fundamental building blocks of the BASIC language. When using the Expression Editor, you can access the BASIC functions via the Function... command on the **Suggest Operand** menu.
- **InfoSphere DataStage BASIC functions**. These are special BASIC functions that are specific to InfoSphere DataStage. These are mostly used in job control routines. InfoSphere DataStage functions begin with DS to distinguish them from general BASIC functions. When using the Expression Editor, you can access the InfoSphere DataStage BASIC functions via the DS Functions... command on the **Suggest Operand** menu.

The following items, although called "functions," are classified as routines and are described under “Routines” on page 125. When using the Expression Editor, they all appear under the DS Routines... command on the **Suggest Operand** menu.

- Transform functions

- Custom UniVerse functions
- ActiveX (OLE) functions

Expressions

An expression is an element of code that defines a value. The word "expression" is used both as a specific part of BASIC syntax, and to describe portions of code that you can enter when defining a job. Areas of IBM InfoSphere DataStage where you can use such expressions are:

- Defining breakpoints in the debugger
- Defining column derivations, key expressions, and constraints in Transformer stages
- Defining a custom transform

In each of these cases the InfoSphere DataStage Expression Editor guides you as to what programming elements you can insert into the expression.

Subroutines

A subroutine is a set of instructions that perform a specific task. Subroutines do not return a value. The word "subroutine" is used both as a specific part of BASIC syntax, but also to refer particularly to before/after subroutines which carry out tasks either before or after a job or an active stage. IBM InfoSphere DataStage has many built-in before/after subroutines, or you can define your own.

Before/after subroutines are included under the general routine classification, as they are accessible from the **Routines** folder in the repository tree by default.

Macros

IBM InfoSphere DataStage has a number of built-in macros. These can be used in expressions, job control routines, and before/after subroutines. The available macros are concerned with ascertaining job status.

When using the Expression Editor, they all appear under the DS Macro... command on the **Suggest Operand** menu.

Precedence Rules

The following precedence rules are applied if there are name conflicts between different operands when working with IBM InfoSphere DataStage programming components:

1. Built-in functions declared in the DSParams file
2. InfoSphere DataStage macros
3. InfoSphere DataStage constants
4. InfoSphere DataStage functions
5. InfoSphere DataStage transforms
6. InfoSphere DataStage routines

These rules ignore the number of arguments involved. For example, if there is a transform with three arguments and a routine of the same name with two arguments, an error is generated if you call the routine because the transform will be found first and the transform expects three arguments.

Working with Routines

When you create, view, or edit a routine, the **Server** Routine dialog box appears. This dialog box has five pages: **General**, **Creator**, **Arguments**, **Code**, and **Dependencies**.

There are five buttons in the Server Routine dialog box. Their availability depends on the action you are performing and the type of routine you are editing.

- **Close**. Closes the **Server** Routine dialog box. If you have any unsaved changes, you are prompted to save them.
- **Save**. Saves the routine.
- **Compile**. Compiles a saved routine. This is available only when there are no outstanding (unsaved) changes.
- **Test...** . Tests a routine. This is available only for routines of type **Transform Function** and **Custom UniVerse Function**. This is because you cannot test before-subroutines and after-subroutines in isolation. This is active only when the routine has compiled or referenced successfully.
- **Help**. Invokes the Help system.

The Server Routine Dialog Box

This section describes the five pages in the Server Routine dialog box.

General Page

The General page is displayed by default. It contains general information about the routine, including:

- **Routine name**. The name of the function or subroutine.
- **Type**. The type of routine. There are three types of routine: **Transform Function**, **Before/After Subroutine**, or **Custom UniVerse Function**.
- **External Catalog Name**. This is only available if you have chosen **Custom UniVerse Function** from the **Type** box. Enter the cataloged name of the external routine.
- **Short description**. An optional brief description of the routine.
- **Long description**. An optional detailed description of the routine.

Creator Page

The Creator page contains information about the creator and version number of the routine, including:

- **Vendor**. The company who created the routine.
- **Author**. The creator of the routine.
- **Version**. The version number of the routine, which is used when the routine is imported. The **Version** field contains a three-part version number, for example, 3.1.1. The first part of this number is an internal number used to check compatibility between the routine and the IBM InfoSphere DataStage system. The second part of this number represents the release number. This number should be incremented when major changes are made to the routine definition or the underlying code. The new release of the routine supersedes any previous release. Any jobs using the routine use the new release. The last part of this number marks intermediate releases when a minor change or fix has taken place.

  If you are creating a routine definition, the first part of the version number is set according to the version of InfoSphere DataStage you are using. You can edit the rest of the number to specify the release level. Click the part of the number you want to change and enter a number directly, or use the arrow button to increase the value.
- **Copyright**. Copyright information.

Arguments Page

The default argument names and whether you can add or delete arguments depends on the type of routine you are editing:

- **Before/After subroutines**. The argument names are InputArg and Error Code. You can edit the argument names and descriptions but you cannot delete or add arguments.

- **Transform Functions and Custom UniVerse Functions**. By default these have one argument called Arg1. You can edit argument names and descriptions and add and delete arguments. There must be at least one argument, but no more than 255.

Code Page

The Code page is used to view or write the code for the routine. The toolbar contains buttons for cutting, copying, pasting, and formatting code, and for activating **Find** (and **Replace**). The main part of this page consists of a multiline text box with scroll bars. For more information about how to use this page, see "Entering Code" on page 130.

Note: This page is not available if you selected **Custom UniVerse Function** on the General page.

Dependencies Page

The Dependencies page allows you to enter any locally or globally cataloged functions or routines that are used in the routine you are defining. This is to ensure that, when you package any jobs using this routine for deployment on another system, all the dependencies will be included in the package. The information required is as follows:

- **Type**. The type of item upon which the routine depends. Choose from the following options:
  - **Local**. Locally cataloged IBM InfoSphere DataStage BASIC functions and subroutines.
  - **Global** . Globally cataloged InfoSphere DataStage BASIC functions and subroutines.
  - **File** . A standard file.
  - **ActiveX.** An ActiveX (OLE) object (not available on UNIX- based systems).
  - **Web Service**. A Web service operation.
- **Name**. The name of the function or routine. The name required varies according to the type of dependency:
  - **Local**. The catalog name.
  - **Global**. The catalog name.
  - **File**. The file name.
  - **ActiveX**. The **Name** entry is actually irrelevant for ActiveX objects. Enter something meaningful to you (ActiveX objects are identified by the **Location** field).
  - **Web Service**. The name of the Web service operation.
- **Location**. The location of the dependency. For a Web service operation, this is a URL. This location can be an absolute path, but it is recommended that you specify a relative path by using the following environment variables:

  %SERVERENGINE% - server engine account directory (normally `C:\IBM\InformationServer\Server\DSEngine`).

  %PROJECT% - Current® project directory.

  %SYSTEM% - System directory on Windows or */usr/lib* on UNIX.

  To browse for the location, double-click to open the Select From Server window. (This window is not available for local cataloged items.) You cannot navigate to the parent directory of an environment variable.

  When browsing for the location of a file on a UNIX server, there is an entry called Root in the **Base Locations** list.

Creating a Routine

To create a new routine:

1. Open the **Server** Routine dialog box in one of these ways:

- Choose **File** → **New** from the main menu or click the **New** button on the toolbar. The New dialog box appears. Click the **Routines** folder and select the **Server Routine** icon.
- Right-click the **Routines** folder in the repository tree and select **New** → **Server Routine** from the shortcut menu.

2. On the General page, enter the name of the function or subroutine in the **Routine name** field. This should not be the same as any BASIC function name.
3. Choose the type of routine you want to create from the **Type** list. There are three options:
   - **Transform Function**. Choose this if you want to create a routine for a Transform definition.
   - **Before/After Subroutine**. Choose this if you want to create a routine for a before-stage or after-stage subroutine or a before-job or after-job subroutine.
   - **Custom UniVerse Function**. Choose this if you want to refer to an external routine, rather than define one in this dialog box. If you choose this, the Code page will not be available.
4. Optionally enter a brief description of the routine in the **Short description** field.
5. Optionally enter a more detailed description of the routine in the **Long description** field.

After this page is complete, you can enter creator information on the Creator page, argument information on the Arguments page, and details of any dependencies on the Dependencies page. You must then enter your code on the Code page.

Entering Code

You can enter or edit code for a routine on the Code page in the **Server** Routine dialog box.

The first field on this page displays the routine name and the argument names. If you want to change these properties, you must edit the fields on the **General** and Arguments pages.

The main part of this page contains a multiline text entry box, in which you must enter your code. To enter code, click in the box and start typing. You can use the following standard Windows edit functions in this text box:

- Delete using the **Del** key
- Cut using **Ctrl-X**
- Copy using **Ctrl-C**
- Paste using **Ctrl-V**
- Go to the end of the line using the **End** key
- Go to the beginning of the line using the **Home** key
- Select text by clicking and dragging or double-clicking

Some of these edit functions are included in a shortcut menu which you can display by right-clicking. You can also cut, copy, and paste code using the buttons in the toolbar.

Your code must only contain BASIC functions and statements supported by IBM InfoSphere DataStage. If you are unsure of the supported functions and statements, or the correct syntax to use, see Chapter 7, “BASIC Programming,” on page 137 for a complete list of supported InfoSphere DataStage BASIC functions.

If NLS is enabled, you can use non-English characters in the following circumstances:

- In comments
- In string data (that is, strings contained in quotation marks)

The use of non-English characters elsewhere causes compilation errors.

If you want to format your code, click the **Format** button on the toolbar.

The last field on this page displays the return statement for the function or subroutine. You cannot edit this field.

Saving Code

When you have finished entering or editing your code, the routine must be saved. A routine cannot be compiled or tested if it has not been saved. To save a routine, click **Save** in the **Server** Routine dialog box. The routine properties (its name, description, number of arguments, and creator information) and the associated code are saved in the repository.

Compiling Code

When you have saved your routine, you must compile it. To compile a routine, click **Compile** in the Server Routine dialog box. If the routine compiles successfully, a message box appears. Click **OK** to acknowledge the message. The routine is marked as "built" in the repository and is available for use. If the routine is a **Transform Function**, it is displayed in the list of available functions when you edit a transform. If the routine is a **Before/After Subroutine**, it is displayed in the list of available subroutines when you edit an Aggregator, Transformer, or supplemental stage, or define job properties. If the routine failed to compile, the errors generated are displayed.

Before you start to investigate the source of the error, you might find it useful to move the Compilation Output window alongside or below the **Server** Routine dialog box, as you need to see both windows to troubleshoot the error.

To troubleshoot the error, double-click the error in the Compilation Output window. IBM InfoSphere DataStage attempts to find the corresponding line of code that caused the error and highlights it in the **Server** Routine dialog box. You must edit the code to remove any incorrect statements or to correct any syntax errors.

If NLS is enabled, watch for multiple question marks in the Compilation Output window. This generally indicates that a character set mapping error has occurred.

When you have modified your code, click **Save** then **Compile**. If necessary, continue to troubleshoot any errors, until the routine compiles successfully.

After the routine is compiled, you can use it in other areas of InfoSphere DataStage or test it. See “Testing a Routine” for more information.

Testing a Routine

Before using a compiled routine, you can test it using the **Test** button in the Server Routine dialog box. The **Test** button is activated when the routine has been successfully compiled.

Note: The **Test** button is not available for a **Before/After Subroutine**. Routines of this type cannot be tested in isolation and must be executed as part of a running job.

When you click **Test**, the Test Routine dialog box appears. This dialog box contains a grid and buttons. The grid has a column for each argument and one for the test result.

You can add and edit rows in the grid to specify the values for different test cases. For more information about using and editing a grid, see *IBM InfoSphere DataStage and QualityStage Designer Client Guide*.

To run a test with a chosen set of values, click anywhere in the row you want to use and click **Run**. If you want to run tests using all the test values, click **Run All**. The **Result...** column is populated as each test is completed.

To see more details for a particular test, double-click the **Result...** cell for the test you are interested in. The Test Output window opens, displaying the full test results. Click **Close** to close this window.

If you want to delete a set of test values, click anywhere in the row you want to remove and press the **Delete** key or choose **Delete row** from the shortcut menu.

When you have finished testing the routine, click **Close** to close the Test Routine dialog box. Any test values you entered are saved when you close the dialog box.

Using Find and Replace

If you want to search the code for specific text, or replace text, you can use **Find** and **Replace**. To start **Find**, click the **Find** button on the Code page toolbar. The Find dialog box appears.

This dialog box has the following fields, options, and buttons:

- **Find what**. Contains the text to search for. Enter appropriate text in this field. If text was highlighted in the code before you chose **Find**, this field displays the highlighted text.
- **Match Case**. Specifies whether to do a case-sensitive search. By default this check box is cleared. Select this check box to do a case-sensitive search.
- **Up** and **Down**. Specifies the direction of search. The default setting is **Down**. Click **Up** to search in the opposite direction.
- **Find Next**. Starts the search. This is unavailable until you specify text to search for. Continue to click **Find Next** until all occurrences of the text have been found.
- **Cancel**. Closes the Find dialog box.
- **Replace...** . Displays the Replace dialog box. For more information, see "Replacing Text."
- **Help**. Invokes the Help system.

Replacing Text

If you want to replace text in your code with an alternative text string, click **Replace...** in the Find dialog box. When you click this button, the Find dialog box changes to the Replace dialog box.

This dialog box has the following fields, options, and buttons:

- **Find what**. Contains the text to search for and replace.
- **Replace with**. Contains the text you want to use in place of the search text.
- **Match Case**. Specifies whether to do a case-sensitive search. By default this check box is cleared. Select this check box to do a case-sensitive search.
- **Up** and **Down**. Specifies the direction of search and replace. The default setting is **Down**. Click **Up** to search in the opposite direction.
- **Find Next**. Starts the search and replace. This button is unavailable until you specify text to search for. Continue to click **Find Next** until all occurrences of the text have been found.
- **Cancel**. Closes the Replace dialog box.
- **Replace**. Replaces the search text with the alternative text.
- **Replace All**. Performs a global replace of all instances of the search text.
- **Help**. Invokes the Help system.

Viewing and Editing a Routine

You can view and edit any user-written functions and subroutines in your project. To view or modify a function or subroutine, select it in the repository tree and do one of the following:

- Choose **Repository** → **Properties**.
- Select **Properties** from the shortcut menu.

- Double-click it in the repository tree.

The **Server** Routine dialog box appears. You can edit any of the fields and options on any of the pages. If you make any changes, you must save, compile, and test the code before closing the **Server** Routine dialog box. See "Saving Code" on page 131 for more information.

Copying a Routine

You can copy an existing routine by selecting it in the repository tree and doing one of the following:

- Choose **Repository** → **Create copy**.
- Select **Create copy** from the shortcut menu.

The routine is copied and a new routine is created under the same folder in the repository tree. By default, the name of the copy is called CopyOf*XXX*, where *XXX* is the name of the chosen routine. An edit box appears allowing you to rename the copy immediately. The new routine must be compiled before it can be used.

Renaming a Routine

You can rename any of the existing routines in the repository. To rename an item, select it in the repository tree and do one of the following:

- Click the routine again. An edit box appears and you can enter a different name or edit the existing one. Save the new name by pressing **Enter** or by clicking outside the edit box.
- Choose **Repository** → **Rename**. An edit box appears and you can enter a different name or edit the existing one. Save the new name by pressing **Enter** or by clicking outside the edit box.
- Select **Rename** from the shortcut menu. An edit box appears and you can enter a different name or edit the existing one. Save the new name by pressing **Enter** or by clicking outside the edit box.
- Double-click the routine. The **Server** Routine dialog box appears and you can edit the **Routine name** field. Click **Save**, then **Close**.

Defining Custom Transforms

Transforms are used in the Transformer stage to convert your data to a format you want to use in the final data mart. Each transform specifies the BASIC function used to convert the data from one type to another. There are a number of built-in transforms supplied with IBM InfoSphere DataStage, which are described in *InfoSphere DataStage Programmer's Guide*.

If the built-in transforms are not suitable or you want a specific transform to act on a specific data element, you can create custom transforms in the Designer client. The advantage of creating a custom transform over just entering the required expression in the Transformer Editor is that, once defined, the transform is available for use from anywhere within the project. It can also be easily exported to other InfoSphere DataStage projects.

To provide even greater flexibility, you can also define your own custom routines and functions from which to build custom transforms. There are three ways of doing this:

- Entering the code within InfoSphere DataStage (using BASIC functions). See "Creating a Routine" on page 129.
- Creating a reference to an externally cataloged routine. See "Creating a Routine" on page 129.
- Importing external ActiveX (OLE) functions. See "Importing External ActiveX (OLE) Functions" on page 135.

To create a custom transform:

1. In the repository project tree, select the **Transforms** folder and do one of the following:
   - Choose **File → New** from the Designer client menu or click the **New** button on the toolbar. The New dialog box appears. Click the **Other** folder and select **Transform**.
   - Right-click and select **New → Other → Transform** from the shortcut menu.

   The Transform dialog box appears. This dialog box has two pages:
   - **General**. Displayed by default. Contains general information about the transform.
   - **Details**. Allows you to specify source and target data elements, the function, and arguments to use.
2. Enter the name of the transform in the **Transform name** field. The name entered here must be unique; no two transforms can have the same name. Also note that the transform should not have the same name as an existing BASIC function; if it does, the function will be called instead of the transform when you run the job. See "Precedence Rules" on page 127 for considerations about component names.
3. Optionally enter a brief description of the transform in the **Short description** field.
4. Optionally enter a detailed description of the transform in the **Long description** field. After this page is complete, you can specify how the data is converted.
5. Click the **Details** tab.
6. Optionally choose the data element you want as the target data element from the **Target data element** list. (Using a target and a source data element allows you to apply a stricter data typing to your transform. See *IBM InfoSphere DataStage and QualityStage Designer Client Guide* for a description of data elements.)
7. Specify the source arguments for the transform in the **Visible Argument** grid. Enter the name of the argument and optionally choose the corresponding data element from the list.

   Use the Expression Editor in the **Definition** field to enter an expression which defines how the transform behaves. The Expression Editor is described in "The IBM InfoSphere DataStage Expression Editor" on page 112. The **Suggest Operand** menu is slightly different when you use the Expression Editor to define custom transforms and offers commands that are useful when defining transforms.

Suggest Operand Menu - Defining Custom Transforms
DS Macro...
DS Function...
DS Constant...
DS Routine...
Transform Argument...
System Variable...
String...
Function...
() Parentheses
If Then Else

8. Click **OK**. The Save Transform As dialog box appears, allowing you to select the folder to save to in the repository tree. You can also rename the transform if required. Click **Save** to save the transform and close the Transform dialog box.

You can then use the new transform from within the Transformer Editor.

Note: If NLS is enabled, avoid using the built-in **Iconv** and **Oconv** functions to map data unless you fully understand the consequences of your actions.

External ActiveX (OLE) Functions

IBM InfoSphere DataStage provides you with the ability to call external ActiveX (OLE) functions which have been installed on the computer where the engine tier resides. These functions can then be used when you define custom transforms.

To use this facility, you need an automation server that exposes functions via the IDispatch interface and which has an associated type library. This can be achieved via a number of development tools, including Visual Basic.

The first step in using external functions is importing them into the repository. The action of importing an external function creates an InfoSphere DataStage routine containing code which calls the external function. The code uses an InfoSphere DataStage BASIC function that accepts only certain data types. These data types are defined in the DSOLETYPES.H file in the dsinclude directory for each project.

Once imported, you can then call the functions when you define a custom transform.

Note: This facility is available only on Windows servers.

Importing External ActiveX (OLE) Functions

To import ActiveX (OLE) functions:

1. In the Designer client, choose **Import → External Function Definitions...**. The Import Transform Functions Definitions wizard opens and prompts you to supply the path name of the file that contains the transforms to import. This is normally a DLL file that must already be installed on the computer where the engine tier resides.
2. Enter or browse for the path name, then click **Next**. The wizard queries the specified DLL file to establish what automation classes it contains and presents these in a list.
3. Select an automation class and click **Next**. The wizard interrogates the automation class to obtain details of the suitable functions it supports. It then displays these.
4. Select the functions that you want to import. Click **Next**. The wizard displays the details of the proposed import.
5. If you are happy with the details, click **Import**. IBM InfoSphere DataStage starts to generate the required routines and displays a progress bar. On completion a summary window opens.
6. Click **Finish** to exit the wizard.

Chapter 7. BASIC Programming

These topics provide a programmer's reference guide for the IBM InfoSphere DataStage BASIC programming language.

The InfoSphere DataStage BASIC described here is the subset of BASIC commands most commonly used in InfoSphere DataStage. You are not limited to the functionality described here, however, you can use the full range of InfoSphere DataStage BASIC commands as described in *IBM InfoSphere DataStage BASIC Reference Guide*, including dynamic arrays. But some areas need care. The main points to watch are as follows:

- Do not use any command, function, statement, or subroutine that requires any user input.
- To stop a running job, use the **DSLogFatal** subroutine. If you use a **Stop** or **Abort** statement, the job might be left in an irrecoverable condition.
- Avoid using the **Print** statement. Use a call to **DSLogInfo** to write to the job log file instead.
- Avoid using the **Execute** statement to execute server engine commands. Use a call to **DSExecute** instead.

The full *IBM InfoSphere DataStage BASIC Reference Guide* is provided in PDF format with InfoSphere DataStage.

Syntax Conventions

The syntax descriptions use the following conventions:

Convention
Usage

Bold Bold type indicates functions, statements, subroutines, options, parenthesis, commas, and so on, that must be input exactly as shown.

Italic Italic indicates variable information that you supply, for example an expression, input string, variable name or list of statements.

[ ] Brackets enclose optional items. Do not enter these brackets.

[ ] Brackets in bold italic typeface must be entered as part of the syntax.

{ Then | Else }
Two keywords or clauses separated by vertical bars and enclosed in braces indicate that you can choose only one option. Do not enter the braces or the vertical bar.

... Three periods indicate that the last item of the syntax can be repeated if required.

@FM Field mark.

@IM Item mark.

@SM Subvalue mark.

@TM Text mark.

@VM Value mark.

The BASIC Language

This section gives an overview of the fundamental components of the IBM InfoSphere DataStage BASIC language. It describes constants, variables, types of data, and how data is combined with arithmetic, strings, relational, and logical operators to form expressions.

Constants

A constant is a value that is fixed during the execution of a program, and can be reused in different contexts. A constant can be:

- A character string
- An empty string
- A numeric string in either floating-point or integer format

ASCII characters 0 and 10, and characters 251 to 255 inclusive cannot be embedded in string constants on non-NLS systems (these characters cannot be used in comments either).

Variables

Variables are used for storing values in memory temporarily. You can then use the values in a series of operations.

You can assign an explicit value to a variable, or assign a value that is the result of operations performed by the program during execution. Variables can change in value during program execution. At the start of program execution, all variables are unassigned. Any attempt to use an unassigned variable produces an error message.

The value of a variable can be:

- Unassigned
- A string
- An integer or floating-point number
- The null value
- A dimensioned array
- A file variable

IBM InfoSphere DataStage provides a set of read-only system variables that store system data such as the current date, time, path name, and so on. These can be accessed from a routine or transform.

Dimensioned Arrays

An array is a multivalued variable accessed from a single name. Each value is an element of the array. IBM InfoSphere DataStage uses two types of dimensioned array:

- One-dimensional arrays, or vectors
- Two-dimensional arrays, or matrices

Vectors have elements stored in memory in a single row. Each element is indexed; that is, it has a sequential number assigned to it. The index of the first element is 1. To specify an element of the vector, use the variable name followed by the index of the element enclosed in parentheses. The index can be a constant or an expression, for example:

```
A(1) ;*specifies the first element of variable A
Cost(n + 1) ;* specifies an expression to calculate the index
```

Matrices have elements stored in several rows. To specify an element of a matrix, you must supply two indices: the row number and the column number. For example, in a matrix with four columns and three rows, the elements are specified using these indices:

```
1,1     1,2     1,3     1,4
2,1     2,2     2,3     2,4
3,1     3,2     3,3     3,4
```

The full specification uses the variable name followed by the indices in parentheses. For example:

```
Obj(3,1)
Widget(7,17)
```

Vectors are treated as matrices with a second dimension of 1. COST(35) and COST(35,1) mean the same.

You define the dimensions of an array with the **Dimension** statement. You can also redimension an array using **Dimension**.

Expressions

An expression defines a value. The value is evaluated at run time. The result can be used as input to a function or be assigned to a variable, and so on. A simple expression can comprise:

- A string or numeric constant, for example, "percent" or "42"
- A variable name
- A function
- A user-defined function

A complex expression can contain a combination of constants, variables, operators, functions, and other expressions.

Functions

A function performs mathematical or string manipulations on the arguments supplied to it, and returns a value. Some functions have no arguments; most have one or more. Arguments are always in parentheses, separated by commas, as shown in this general syntax:

```
FunctionName (argument, argument)
```

An expression can contain a function. An argument to a function can be an expression that includes a function. Functions can perform tasks:

- On numeric strings, such as calculating the sine of an angle passed as an argument (**Sin** function)
- On character strings, such as deleting surplus blank spaces and tabs (**Trim** function)

Transform functions in IBM InfoSphere DataStage must have at least one argument that contains the input value to be transformed. Subsequent, optional, arguments can be used by a transform definition to select a particular path through the transform function, if required. This means that a single function can encapsulate the logic for several related transforms. The transform function must return the transformed value using a **Return (*value*)** statement.

Statements

Statements are used for:

- Changing program control. Statements are executed in the order in which they are entered, unless a control statement changes the order by, for example, calling a subroutine, or defining a loop.
- Assigning a value to a variable.
- Specifying the value of a constant.
- Adding comments to programs.

Statement Labels

A statement label is a unique identifier for a line of code. A statement label consists of a string of up to 64 characters followed by a colon. The string can contain alphanumeric characters, periods, dollar signs, and percent signs. Statement labels are case-sensitive. A statement label can be put either in front of a statement or on its own line.

Subroutines

A subroutine is a self-contained set of instructions that perform a specific task. A subroutine can take two forms:

- An embedded subroutine is contained within the program and is accessed with a **GoSub** statement.
- An external subroutine is stored in a separate file and is accessed with a **Call** statement.

In general terms, use an embedded subroutine for code that you want to call many times from the same program; use an external subroutine for code that you want to call from many different programs.

There are a number of BASIC subroutines that are specific to IBM InfoSphere DataStage. Their names begin with DS and they are described in "Special IBM InfoSphere DataStage BASIC Subroutines."

InfoSphere DataStage is also supplied with a number of before/after subroutines, for running before or after a job or an active stage. You can define your own before/after subroutines using the Designer client. Before/after subroutines must have two arguments. The first contains the value a user enters when the subroutine is called from a job or stage; the second is the subroutine's reply code. The reply code is 0 if there was no error. Any other value indicates the job was stopped.

Special IBM InfoSphere DataStage BASIC Subroutines

InfoSphere DataStage provides some special InfoSphere DataStage subroutines for use in a before/after subroutines or custom transforms. You can:

- Log events in the job's log file using **DSLogInfo**, **DSLogWarn**, **DSLogFatal**, and **DSTransformError**
- Execute DOS or server engine commands using **DSExecute**

All the subroutines are called using the **Call** statement.

Operators

An operator performs an operation on one or more expressions (the operands). Operators are divided into these categories:

- Arithmetic operators
- String operators for:
  - Concatenating strings with **Cats** or **:**
  - Extracting substrings with*[ ]*
- Relational operators
- Pattern matching operators
- **If** operator
- Logical operators
- Assignment operators

Arithmetic Operators

Arithmetic operators combine operands by adding, subtracting, and so on. The resulting expressions can be further combined with other expressions. Operands must be numeric expressions. Nonnumeric expressions are treated as 0 and generate a runtime warning. A character string variable containing only numeric characters counts as a numeric expression. For example, the following expression results in the value 66:

```
"22" + 44
```

This table lists the arithmetic operators in order of evaluation:

Operator	Operation	Example
-	Negation	-x
^	Exponentiation	x ^ y
*	Multiplication	x * y
/	Division	x / y
+	Addition	x + y
-	Subtraction	x - y

You can change the order of evaluation using parentheses. Expressions enclosed in parentheses are evaluated before those outside parentheses.

For example, this expression is evaluated as 112 + 6 + 2, or 120:

```
(14 * 8) + 12 / 2 + 2
```

This expression is evaluated as 14 * 20 / 4, or 280 / 4, or 70:

```
14 * (8 + 12) / (2 + 2)
```

The result of any arithmetic operation involving the null value is a null value.

Concatenating Strings

The concatenation operator, **:** or **Cats**, links string expressions to form compound string expressions. For example, if *x* has a value of Tarzan, this expression:

```
"Hello. " : "My Name is " : X : ". What's yours?"
```

evaluates to:

```
"Hello. My name is Tarzan. What's yours?"
```

Multiple concatenation operations are normally performed from left to right. You can change the order of evaluation using parentheses. Parenthetical expressions are evaluated before operations outside the parentheses.

Numeric operands in concatenated expressions are considered to be string values. Arithmetic operators have higher precedence than the concatenation operator. For example:

```
"There are " : "2" + "2" : "3" : " windows."
```

has the value:

```
"There are 43 windows."
```

The result of any string operation involving the null value is a null value. But if the null value is referenced as a character string containing only the null value (that is, as the string CHAR(128)), it is treated as character string data. For example, this expression evaluates to null:

```
"A" : @NULL ;*concatenate A with @NULL system variable
```

But this expression:

```
"A" : @NULL.STR ;*concatenate A with @NULLSTR system variable
```

evaluates to "A<CHAR128>".

Extracting Substrings

A substring is a string within a string. For example, tab and able are both substrings of table. You can use the*[ ]* operator to specify a substring using this syntax:

```
string[ [ start, ] length ]
```

string is the string containing the substring.

start is a number specifying where the substring starts. The first character of *string* counts as 1. If *start* is 0 or a negative number, the starting position is assumed to be 1. If *start* is omitted, the starting position is calculated according to the following formula:

```
string.length - substring.length + 1
```

- **Trailing Substrings**. You can specify a trailing substring by omitting *start* from the syntax. For example, this specification:

  ```
  "1234567890" [5]
  ```

  returns the substring:

  ```
  67890
  ```

- **Delimited Substrings**. You can extract a delimited substring using this syntax:

  ```
  string [ delimiter, instance, fields ]
  ```

  – *string* is the string containing the substring.
  – *delimiter* specifies the character that delimits the substring.
  – *instance* specifies the instance of delimiter where the extraction is to start.
  – *fields* specifies the number of fields to extract.

  The delimiters that mark the start and end of the extraction are not returned, but if you extract more than one string, any interim delimiters are returned. This syntax works the same way as the **Field** function.

- **Assigning a Substring to a Variable**. All substring syntaxes can be used with the = operator to replace the value normally returned by the *[ ]* operator with the value assigned to the variable. For example:

  ```
  A="12345"
  A[3]=1212
  ```

  returns the result 121212.

  This syntax works the same way as the **FieldStore** function.

Relational Operators

Relational operators compare strings or other data. The result of the comparison is either true (1) or false (0). This table shows the relational operators you can use:

Operator	Relation	Example
Eq or =	Equality	X = Y
Ne or # or >< or <>	Inequality	X # Y, X <> Y
Lt or <	Less than	X < Y
Gt or >	Greater than	X > Y
Le or <= or =< or #>	Less than or equal to	X <= Y
Ge or >= or => or #<	Greater than or equal to	X >= Y

Arithmetic operations are performed before any relational operators in an expression. For example, the expression:

```
X + Y < (T - 1) / Z
```

is true if the value of X plus Y is less than the value of T minus 1 divided by Z.

Strings are compared character by character. The string with the higher character code is considered to be greater. If all the character codes are the same, the strings are considered equal.

A space is evaluated as less than 0. A string with leading or trailing blanks is considered greater than the same string without the blanks.

An empty string is always compared as a character string. It does not equal numeric 0.

If two strings contain numeric characters they are compared numerically. For example:

```
"22" < "44" '
```

returns true.

Take care if you use exponentiation notation. For example:

```
"23" > "2E1"
```

returns true.

Here are some examples of true comparisons in ASCII 7-bit with standard collating conventions:

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"CL " > "CL"
"kg" > "KG"
"SMYTH" < "SMYTHE"
B$ < "9/14/99" ;* where B$ = "8/14/99"
```

You cannot use relational operators to test for a null value. Use the **IsNull** function instead.

Pattern Matching Operators

Pattern matching operators compare a string with a format pattern. If NLS is enabled, the result of a match operation depends on the current locale setting of the Ctype and Numeric conventions. Pattern matching operators have the following syntax:

```
string Match pattern
```

string is the string to be compared. If *string* is a null value, the match is false and 0 is returned.

pattern is the format pattern, and can be one of the following codes:

This code...
Matches this type of string...

... Zero or more characters of any type.

0X Zero or more characters of any type.

*n***X** *n* characters of any type.

0A Zero or more alphabetic characters.

*n***A** *n* alphabetic characters.

0N Zero or more numeric characters.

*n***N** *n* numeric characters.

'*string* **'**
Exact text enclosed in double or single quotation marks.

You can specify a negative match by preceding the code with ~ (tilde). For example, ~ 4A matches a string that does not contain four alphabetic characters. If *n* is longer than nine digits, it is used as a literal string.

If *string* matches *pattern*, the comparison returns 1, otherwise it returns 0.

You can specify multiple patterns by separating them with value marks. For example, the following expression is true if the address is either 16 alphabetic characters or 4 numeric characters followed by 12 alphabetic characters; otherwise, it is false:

```
address Matches "16A": CHAR(253): "4N12A"
```

An empty string matches the following patterns: "0A", "0X", "0N", "...", "", '', or \\.

If Operators

An **If** operator assigns a value that meets the specified conditions. It has the following syntax:

```
variable = If condition Then expression Else expression
```

variable is the variable to assign.

If *condition* defines the condition that determines which value to assign.

Then *expression* defines the value to assign if *condition* is true.

Else *expression* defines the value to assign if *condition* is false.

The **If** operator is the only form of **If...Then...Else** construction that can be used in an expression.

Note that the **Else** clause is required in the following examples:

```
* Return A or B depending on value in Column1:
If Column1 > 100 Then "A" Else "B"
* Add 1 or 2 to value in Column2 depending on what's in
* Column3, and return it:
Column2 + (If Column3 Matches "A..." Then 1 Else 2)
```

Logical Operators

Numeric data, string data, and the null value can function as logical data:

- The numeric value 0, is false; all other numeric values are true.
- An empty string is false; all other character strings are true.
- The SQL null value is neither true nor false. It has the logical value of null.

Logical operators test for these conditions. The logical operators available are:

- **And** (or the equivalent **&**)
- **Or** (or the equivalent **!**)
- **Not** (inverts a logical value)

These are the factors that determine operator precedence in logical operations:

- Arithmetic and relational operations take precedence over logical operations.
- Logical operations are evaluated from left to right.
- **And** statements and **Or** statements have equal precedence.
- In **If...Then...Else** clauses, the logical value null takes the false action.

Assignment Operators

Assignment operators assign values to variables. This table shows the assignment operators and their uses:

Operator	Syntax	Description
=	*variable* = *expression*	Assigns the value of *expression* to *variable*.
+=	*variable* += *expression*	Adds the value of *expression* to the value of *variable* and reassigns the result to *variable*.
-=	*variable* - = *expression*	Subtracts the value of *expression* from the value of *variable* and reassigns the result to variable.
:=	*variable* := *expression*	Concatenates the value of *variable* and the value of *expression* and reassigns the result to *variable*.

This example shows a sequence of operations on the same variable. The first statement assigns the value 5 to the variable X.

```
X = 5
```

The next statement adds 5 to the value of X, and is equivalent to X = X + 5. The value of X is now 10.

```
X += 5
```

The final statement subtracts 3 from the value of X, and is equivalent to X = X - 3. The value of X is now 7.

```
X -= 3
```

This example concatenates a string with the variable and is equivalent to X = X:Y. If the value of X is `con', and the value of Y is `catenate':

```
X := Y
```

the new value of X is `concatenate'.

Data Types in BASIC Functions and Statements

You do not need to specify data types in functions and statements. All data is stored internally as character strings, and data types are determined at run time, according to their context. There are four main types of data:

- **Character strings**. These can represent alphabetic, numeric, or alphanumeric data such as an address. String length is limited only by available memory.
- **Numeric data**. This is stored as either floating-point numbers, or as integers. On most systems the range is 10^{-307} through 10^{+307} with 15 decimal digits of precision.
- **The null value**. This represents data whose value is unknown, as defined by SQL.
- **File variables**. These are assigned by the **OpenSeq** statement, and cannot be manipulated or formatted in any way.

Empty BASIC Strings and Null Values

An empty string is a character string of zero length. It represents known data that has no value. Specify an empty string with two adjacent double quotation marks, single quotation marks, or backslashes. For example:

```
' ' or " " or \\
```

The null value represents data whose value is unknown, as defined by SQL.

The null value is represented externally, as required, by a character string consisting of the single byte Char(128). At run time it is assigned a data type of null. Programs can reference the null value using the system variable @NULL. To test if a value is the null value, use the **IsNull** function.

If you input a null value to a function or other operation, a null value is always returned. For example, if you concatenate a string value with an empty string, the string value is returned, but if you concatenate a string value with the null value, null is returned:

```
A = @NULL
B = ""
C = "JONES"
X = C:B
Y = C:A
```

The resulting value of X is "JONES", but Y is a null value.

Fields

In IBM InfoSphere DataStage functions such as **Field** or **FieldStore**, you can define fields by specifying delimited substrings. What constitutes a field is determined as follows:

- Any substring followed by a delimiter is a field.
- If a string starts with a delimiter, InfoSphere DataStage assumes there is a field containing an empty string in front of the delimiter.
- If a trailing substring does not end with a delimiter, InfoSphere DataStage assumes there is one.

For example, if you use the string ABC with a colon as a delimiter, InfoSphere DataStage generates either three or four fields, as follows:

Example	Number of Fields	Explanation
A : B : C :	3	Each field ends with a delimiter.
A : B : C	3	InfoSphere DataStage assumes the final delimiter.
: A : B : C :	4	InfoSphere DataStage assumes a field containing an empty string before the first delimiter.
: A : B : C	4	InfoSphere DataStage assumes a field containing an empty string before the first delimiter, and assumes a final delimiter.

Reserved Words

These words are reserved and should not be used as variable names in a transform or routine:

- And
- Cat
- Else
- End
- Eq
- Ge

- Get
- Go
- GoSub
- GoTo
- Gt
- If
- Include
- Le
- Locked
- Lt
- Match
- Matches
- Ne
- Next
- Or
- Rem
- Remove
- Repeat
- Then
- Until
- While

Source Code and Object Code

Source code is the original input form of the routine the programmer writes.

Object code is the compiled output that IBM InfoSphere DataStage calls as a subroutine or a function.

A source line has the following syntax:

```
[ label: ] statement [ ; statement ] ...<Return>
```

A source line can begin with a statement label. It always ends with a **Return**.

Special Characters

The following characters have a special meaning in transforms and routines. Their use is restricted in numeric and string constants. Also note that ASCII characters 0 through 10 and 251 through 255 should not be embedded in string constants.

Character	Permitted Use
Space	Used in string constants, or for formatting source code.
Tab	Used in string constants, or for formatting source code.
=	Used to indicate the equality or assignment operators.
+	Plus. Used to indicate the addition operator or unary plus.
-	Minus. Used to indicate the subtraction operator or unary minus.
*	Asterisk. Used to indicate the multiplication operator or a comment in source code.

\ Backslash. Used for quoting strings.

/ Slash. Used to indicate the division operator.

^ Up-arrow. Used to indicate the exponentiation operator.

( ) Parentheses. Used to enclose arguments in functions or matrix dimensions.

Hash. Used to indicate the not equal operator.

$ Dollar sign. Allowed in variable names and statement labels, but not allowed in numeric constants.

[ ] Brackets. Used to indicate the substring extraction operator, and to enclose certain expressions.

, Comma. Used to separate arguments in functions and subroutines or matrix dimensions. Not permitted in numeric constants.

. Period. Used to indicate a decimal point in numeric constants.

"" Double quotation marks. Used to quote strings.

" Single quotation marks. Used to quote strings.

: Colon: Used to indicate the concatenation operator, or the end of a statement label.

; Semicolon. Used to indicate the end of a statement if you want to include a comment on the same line.

& Ampersand. Used to indicate the And relational operator.

< Left angle bracket. Used to indicate the less than operator.

> Right angle bracket. Used to indicate the greater than operator.

@ At sign. Reserved for use in system variables.

System Variables

IBM InfoSphere DataStage provides a set of variables containing useful system information that you can access from a transform or routine. System variables are read-only.

Name Description

@DATE
The internal date when the program started. See the **Date** function.

@DAY The day of the month extracted from the value in @DATE.

@FALSE
The compiler replaces the value with 0.

@FM A field mark, Char(254).

@IM An item mark, Char(255).

@INROWNUM
Input row counter. For use in constraints and derivations in Transformer stages.

@OUTROWNUM
Output row counter (per link). For use in derivations in Transformer stages.

@LOGNAME
The user login name.

@MONTH
The current extracted from the value in @DATE.

@NULL
The null value.

@NULL.STR
The internal representation of the null value, Char(128).

@PATH
The path name of the current InfoSphere DataStage project.

@SCHEMA
The schema name of the current InfoSphere DataStage project.

@SM A subvalue mark, Char(252).

@SYSTEM. RETURN.CODE
Status codes returned by system processes or commands.

@TIME
The internal time when the program started. See the Time function.

@TM A text mark, Char(251).

@TRUE
The compiler replaces the value with 1.

@USERNO
The user number.

@VM A value mark, Char(253).

@WHO
The name of the current InfoSphere DataStage project directory.

@YEAR
The current year extracted from @DATE.

BASIC Functions and Statements

Compiler Directives

Compiler directives are statements that determine how a routine or transform is compiled.

To do this...
Use this...

Add or replace an identifier
$Define Statement

Remove an identifier
$Undefine Statement

Specify conditional compilation
$IfDef and $IfNDef Statements

Include another program
$Include Statement

Declaration

These statements declare arrays, functions, and subroutines for use in routines.

To do this...
Use this...

Define a storage area in memory
Common Statement

Define a user-written function
Deffun Statement

Declare the name and dimensions of an array variable
Dimension Statement

Identify an internal subroutine
Subroutine Statement

Job Control

These functions can be used in a job control routine, which is defined as part of a job's properties and allows other jobs to be run and controlled from the first job. Some of the functions can also be used for getting status information about the current job; these are useful in active stage expressions and before- and after-stage subroutines.

To do this...
Use this...

Specify the job you want to control
DSAttachJob

Set parameters for the job you want to control
DSSetParam

Set limits for the job you want to control
DSSetJobLimit

Request that a job is run
DSRunJob

Wait for a called job to finish
DSWaitForJob

Get information about the current project
DSGetCustInfo

Get information about the controlled job or current job
DSGetProjectInfo

Get information about a stage in the controlled job or current job
DSGetJobInfo

Get information about a link in a controlled job or current job
DSGetStageInfo

Get information about a controlled job's parameters
DSGetLinkInfo

Get the log event from the job log
DSGetParamInfo

Get a number of log events on the specified subject from the job log
DSGetLogEntry

Get a list of log event IDs for a given run of a job invocation
DSGetLogEventIds

Get the newest log event, of a specified type, from the job log
DSGetLogSummary

Log an event to the job log of a different job
DSLogEvent

Log a fatal error message in a job's log file and abort the job
DSLogFatal

Log an information message in a job's log file
DSLogInfo

Put an info message in the job log of a job controlling current job
DSLogToController

Log a warning message in a job's log file
DSLogWarn

Generate a string describing the complete status of a valid attached job
DSMakeJobReport

Insert arguments into the message template
DSMakeMsg

Ensure a job is in the correct state to be run or validated
DSPrepareJob

Interface to system send mail facility
DSSendMail

Log a warning message to a job log file
DSTransformError

Convert a job control status or error code into an explanatory text message
DSTranslateCode

Suspend a job until a named file either exists or does not exist
DSWaitForFile

Check if a BASIC routine is cataloged, either in VOC as a callable item or in the catalog space
DSCheckRoutine

Execute a DOS or server engine command from a before/after subroutine
DSExecute

Stop a controlled job
DSStopJob

Return a job handle previously obtained from DSAttachJob
DSDetachJob

Set a status message for a job to return as a termination message when it finishes
DSSetUserStatus

Specify whether a job generates operational metadata as it runs (overrides the default setting for the project)
DSSetGenerateOpMetaData

Program Control

These statements control program flow by direct program execution through loops, subroutines, and so on.

To do this...
Use this...

Start a set of Case statements
Begin Case (see Case Statement)

Specify conditions for program flow
Case (see Case Statement)

End a set of Case statements
End Case (see Case Statement)

End a program or block of statements
End Statement

Call an external subroutine
Call Statement

Call an internal subroutine
GoSub Statement

Specify a condition to call an internal subroutine
On...GoSub Statements

Return from an internal or external subroutine
Return Statement

Define the start of a For...Next loop
For (see For...Next Statements)

Define the end of a For...Next loop
Next (see For...Next Statements)

Go to the next iteration of a loop
Continue (see For...Next Statements)

Create a loop
Loop...Repeat Statements

Define conditions for a loop to stop
While, Until (see For...Next Statements)

Exit a loop
Exit (see For...Next Statements)

Branch to a statement unconditionally
GoTo Statement

Branch to a statement conditionally
On...GoTo Statement

Specify conditions for program flow
If...Then...Else Operator

Sequential File Processing

These statements and functions are used to open, read, and close files for sequential processing.

To do this...
Use this...

Open a file for sequential processing
OpenSeq Statement

Read a line from a file opened with OpenSeq
ReadSeq

Write a line to a file opened with OpenSeq
WriteSeq Function

Write a line to a file opened with OpenSeq saved to disk
WriteSeqF Function

Truncate a file opened with OpenSeq
WEOFSeq Function

Close a file opened with OpenSeq
CloseSeq Statement

Find the status of a file opened with OpenSeq
Status Function

String Verification and Formatting

These functions carry out string formatting tasks.

To do this...
Use this...

Check if a string is alphabetic
Alpha Function

Verify with a 16-bit checksum
Checksum Function

Verify with a 32-bit cyclic redundancy check code
CRC32 Function

Enclose a string in double quotation marks
DQuote Function

Enclose a string in single quotation marks
SQuote Function

Analyze a string phonetically
Soundex Function

Convert a string to uppercase
UpCase Function

Convert a string to lowercase
DownCase Function

Replace specified characters in a variable
Convert Function

Replace specified characters in a string
Convert Statement

Replace or delete characters in a string
Exchange Function

Compare two strings for equality
Compare Function

Calculate the number of characters in a string
Len Function

Calculate the length of a string in display positions
LenDP Function

Trim surplus white space from a string
Trim Function TrimB Function TrimF Function

Make a string consisting of spaces only
Space Function

Substring Extraction and Formatting

You can extract and manipulate substrings and fields using these functions.

To do this...
Use this...

Find the starting column of a substring
Index Function

Replace one or more instances of a substring
Change Function

Return the column position before or after a substring
Col1 Function Col2 Function,

Count the number of times a substring occurs in a string
Count Function

Count delimited substrings in a string
DCount Function

Replace one or more instances of a substring
Ereplace Function

Return a delimited substring
Field Function

Replace, delete, or insert substrings in a string
FieldStore Function

Fold strings to create substrings
Fold Function

Fold strings to create substrings using character display positions
FoldDP Function

Extract the first *n* characters of a string
Left Function

Extract the last *n* characters of a string
Right Function

Find a substring that matches a pattern
MatchField Function

Repeat a string to create a new string
Str Function

Searches a dynamic array for an expression
LOCATE Statement

Data Conversion

These functions perform numeric and character conversions.

To do this...
Use this...

Convert ASCII code values to their EBCDIC equivalents
Ebcdic Function

Convert EBCDIC code values to their ASCII equivalents
Ascii Function

Convert an ASCII code value to its character equivalent
Char Function

Convert an ASCII character to its code value
Seq Function

Convert hexadecimal values to decimal
Xtd Function

Convert decimal values to hexadecimal
Dtx Function

Convert numeric value to floating point with specified precision
FIX Function

Convert numeric to floating point without loss of accuracy
REAL Function

Generate a single character in Unicode format
UniChar Function

Convert a Unicode character to its equivalent decimal value
UniSeq Function

Data Formatting

These functions can be used to format data into times, dates, monetary amounts, and so on.

To do this...
Use this...

Convert data for output
Oconv Function

Convert data on input
Iconv Function

Format data for output
Fmt Function

Format data by display position
"FmtDP Function" on page 215

Locale Functions

These functions are used to set or identify the current locale.

To do this...
Use this...

Set a locale
SetLocale

Get a locale
GetLocale Function

$Define Statement

Defines identifiers that control program compilation or supplies replacement text for an identifier. Not available in expressions.

Syntax

```
$Define identifier [replacement.text]
```

identifier is the symbol to be defined. It can be any valid identifier.

replacement.text is a string of characters that the compiler uses to replace *identifier* everywhere it appears in the program containing the **$Define** statement.

Remarks

Enter one blank to separate *identifier* from *replacement.text*. Any further blanks are taken as part of *replacement.text*. End *replacement.text* with a newline. Do not include comments after *replacement.text* or they are included as part of the replacement text.

Examples

This example shows how **$Define** can be used at compile time to determine whether a routine operates in a debugging mode, and how **$IfDef** and **$IfNDef** are used to control program flow accordingly:

```
* Set next line to $UnDefine to switch off debugging code
$Define DebugMode
...
$IfDef DebugMode
* In debugging mode, log each time through this routine.
Call DSLogInfo("Transform entered,arg1 = ":Arg1, "Test")
$EndIf
```

This example shows how **$Define** can be used to replace program text with a symbolic identifier:

```
* Give a symbolic name to the last 3 characters of the
* transform routine's incoming argument.
$Define NameSuffix Arg1[3]
   ...
If NameSuffix = "X27" Then
* Action is based on particular value in last 3 characters.
...End
```

$IfDef and $IfNDef Statements

Tests an identifier to see if it is defined or not defined. Not available in expressions.

Syntax

```
{$IfDef | IfNDef} identifier   [ statements ]
$Else   [ statements ]
$EndIf
```

identifier is the identifier to test for.

$Else specifies alternative statements to execute.

$EndIf ends the conditional compilation block.

Remarks

With **$IfDef**, if *identifier* is defined by a prior **$Define** statement, all the program source lines appearing between the **$IfDef** statement and the closing **$EndIf** statement are compiled. With **$IfNDef**, the lines are compiled if the identifier is not defined. **$IfDef** and **$IfNDef** statements can be nested up to 10 deep.

Example

This example shows how **$Define** can be used at compile time to determine whether a routine operates in a debugging mode, and how **$IfDef** and **$IfNDef** are used to control program flow accordingly:

```
* Set next line to $UnDefine to switch off debugging code
$Define DebugMode
...
$IfDef DebugMode
* In debugging mode, log each time through this routine.
Call DSLogInfo("Transform entered,arg1 = ":Arg1, "Test")
$EndIf
```

$Include Statement

Inserts source code contained in a separate file and compiles it along with the main program. Not available in expressions.

Syntax

$Include *program*

Remarks

The included file must be in the project subdirectory DSU_BP. You can nest **$Include** statements.

$Undefine Statement

Removes an identifier that was set using the **$Define** statement. If no *identifier* is set, **$Undefine** has no effect. Not available in expressions.

Syntax

$Undefine *identifier*

[] Operator

Syntax

Extracts a substring from a character string. The second syntax acts like the **Field** function. The square brackets of the **[ ]** operator are shown in bold italics in the syntax and must be entered.

string [***[*** *start*,***]*** *length*]
*string****[*** *delimiter*, *instance*, *repeats* ***]***

string is the character string. If *string* is a null value, the extracted value is also null.

start is a number that defines the starting position of the first character in the substring. A value of 0 or a negative number is assumed to be 1. If you specify a starting position after the end of *string*, an empty string is returned.

length is the number of characters in the substring. If you specify 0 or a negative number, an empty string is returned. If you specify more characters than there are left between start and the end of *string*, the value returned contains only the number of characters left in *string*.

delimiter is a character that delimits the start and end of the substring. If *delimiter* is not found in *string*, an empty string is returned unless *instance* is 1, in which case *string* is returned.

instance specifies which instance of the delimiter marks the end of the substring. A value of less than 1 is assumed to be 1.

repeat specifies the number of times the extraction is repeated on the string. A value of less than 1 is assumed to be 1. The delimiter is returned along with the successive substrings.

Remarks

You can specify a substring consisting of the last *n* characters of a string by using the first syntax and omitting *start*.

Examples

In the following example (using the second syntax) the fourth # is the terminator of the substring to be extracted, and one field is extracted:

```
A = "###DHHH#KK"
B = A["#",4,1]
```

The result is B equals DHHH.

The following syntaxes specify substrings that start at character position 1:

```
expression [ 0, length ]
expression [ -1, length ]
```

The following example specifies a substring of the last five characters:

```
"1234567890" [5]
```

The result is 67890.

All substring syntaxes can be used with the assignment operator (=). The new value assigned to the variable replaces the substring specified by the **[ ]** operator. This usage is not available in expressions. For example:

```
A = '12345'
A[3] = 1212
```

The result is A equals 121212.

Because no length argument was specified, A[3] replaces the last three characters of A, (345) with the newly assigned value for that substring (1212).

---

* Statement

Inserts a comment in a program.

Syntax

```
* [comment.text]
```

Remarks

A comment can appear anywhere in a program, except in replacement text for an identifier (see the **$Define** statement). Each full comment line must start with an asterisk (*). If you put a comment at the end of a line containing an executable statement, you must put a semicolon (;) before the asterisk.

Example

This example contains both an inline comment and a whole-line comment:

```
MyVar = @Null    ;* sets variable to null value
If IsNull(MyVar * 10) Then
* Will be true since any arithmetic involving a null value
* just results in a null value.
End
```

Abs Function

Returns the absolute (unsigned) value of a number.

Syntax

Abs (*number*)

number is the number or expression you want to evaluate.

Remarks

A useful way to remove plus or minus signs from a string. For example, if *number* is either -6 or +6, **Abs** returns 6. If *number* is a null value, a null value is returned.

Example

This example uses the **Abs** function to compute the absolute value of a number:

```
AbsValue = Abs(12.34)            ;* returns 12.34
AbsValue = Abs(-12.34)            ;* returns 12.34
```

Alpha Function

Checks if a string is alphabetic. If NLS is enabled, the result of this function is dependent on the current locale setting of the Ctype convention.

Syntax

Alpha (*string*)

string is the string or expression you want to evaluate.

Remarks

Alphabetic strings contain only the characters a through z or A through Z. **Alpha** returns 1 if the string is alphabetic, a null value if the string is a null value, and 0 otherwise.

Examples

These examples show how to check that a string contains only alphabetic characters:

```
Column1 = "ABcdEF%"
* the "%" character is non-alpha
Column2 = (If Alpha(Column1) Then "A" Else "B")
* Column2 set to "B"
Column1 = ""
* note that the empty string is non-alpha
Column2 = (If Alpha(Column1) Then "A" Else "B")
* Column2 set to "B"
```

Ascii Function

Converts the values of characters in a string from EBCDIC to ASCII format.

Syntax

Ascii (*string*)

string is the string or expression that you want to convert. If *string* is a null value, a null value is returned.

Remarks

The **Ascii** and **Ebcdic** functions perform complementary operations.

Note: If NLS is enabled, this function might return data that is not recognized by the current character set map.

Example

This example shows the **Ascii** function being used to compare a string of EBCDIC bytes:

```
EbcdicStr = Char(193):Char(241)             ;* letter A digit 1 in EBCDIC
AsciiStr = Ascii(EbcdicStr)                  ;* convert EBCDIC to ASCII
If AsciiStr = "A1" Then            ;* compare with ASCII constant
...                                         ;* ... this branch is taken
EndIf
```

Assignment Statement

The assignment statements are =, +=, -=, and :=. They assign values to variables. Not available in expressions.

s

Syntax

```
variable = value
variable += value
variable -= value
variable := value
```

value is the value you want to assign. It can be any constant or expression, including a null value.

Remarks

= assigns *value* to *variable*.

+= adds *value* to *variable*.

-= subtracts *value* from *variable*.

:= concatenates *value* to the end of *variable*.

To assign a null value to a variable, use this syntax:

```
variable = @NULL
```

To assign a character string containing only the character used to represent the null value to a variable, use this syntax:

```
variable = @NULL.STR
```

Bit functions

The **Bit** functions are **BitAnd**, **BitOr**, **BitNot**, **BitSet**, **BitReset**, **BitTest**, and **BitXOr**. They perform bitwise operations on integers.

Syntax

```
BitAnd | BitOr | BitXOr (integer1, integer2)
BitSet | BitReset | BitTest (integer, bit.number)
BitNot (integer [,bit.number])
```

integer1 and *integer2* are integers to be compared. If either *integer* is a null value, a null value is returned. Decimal places are truncated before the evaluation.

integer is the integer to be evaluated. If *integer* is a null value, a null value is returned. Decimal places are truncated before the evaluation.

bit.number is the number of the bit to act on. Bits are counted from right to left starting with 0. If *bit.number* is a null value, the program fails with a runtime error.

Remarks

The **Bit** functions operate on a 32-bit twos-complement word. Do not use these functions if you want your code to be portable, as the top bit setting might differ on other hardware.

BitAnd compares two integers bit by bit. For each bit, it returns bit 1 if both bits are 1; otherwise it returns bit 0.

BitOr compares two integers bit by bit. For each bit, it returns bit 1, if either or both bits is 1; otherwise it returns bit 0.

BitXOr compares two integers bit by bit. For each bit, it returns bit 1 if only one of the two bits is 1; otherwise it returns bit 0.

BitTest tests if the specified bit is set. It returns 1 if the bit is set; 0 if it is not.

BitNot inverts the bits in an integer, that is, changes bit 1 to bit 0, and vice versa. If *bit.number* is specified, that bit is inverted; otherwise all bits are inverted.

BitSet sets the specified bit to 1. If it is already 1, it is not changed.

BitReset resets the specified bit to 0. If it is already 0, it is not changed.

Examples

BitAnd

```
Result = BitAnd(6, 12)         ;* Result is 4
*  (bin) (dec) BitAnd (bin) (dec) gives (bin) (dec)
*   110     6              1100     12           100    4
```

BitNot

```
Result = BitNot(6)             ;* Result is -7
Result = BitNot(15, 0)         ;* Result is 14
Result = BitNot(15, 1)         ;* Result is 13
Result = BitNot(15, 2)         ;* Result is 11
* (bin) (dec) BitNot  bit#  gives (bin) (dec)
*  110   6     (all)              1...1001    7
* 1111   15      0                   1110   14
* 1111   15      1                   1101   13
* 1111   15      2                   1011   11
```

BitOr

```
Result = BitOr(6, 12)          ;* Result is 14
*  (bin) (dec) BitOr (bin) (dec) gives (bin) (dec)
*   110     6                 1100    12         1110 14
```

BitReset

```
Result = BitReset(29, 0)               ;* Result is 28
Result = BitReset(29, 3)               ;* Result is 21
Result = BitReset(2, 1)               ;* Result is 0
Result = BitReset(2, 0)                ;* Result is 2
*  (bin) (dec) BitReset bit#  gives (bin) (dec)
*  11101    29              0         11100    28
*  11101    29              3         10101    21
*    10      2               1            00     0
*    10      2               0            10     2
```

BitSet

```
Result = BitSet(20, 0)               ;* Result is 21
Result = BitSet(20, 3)               ;* Result is 28
Result = BitSet(2, 0)               ;* Result is 3
Result = BitSet(2, 1)               ;* Result is 2
*  (bin) (dec) BitReset bit#  gives (bin) (dec)
*  10100   20             0        10101    21
*  10100   20             2        11100    28
*     10    2             0           11     3
*     10    2             1           10     2
```

BitTest

```
Result = BitTest(11, 0)        ;* Result is 1
Result = BitTest(11, 1)        ;* Result is 1
Result = BitTest(11, 2)        ;* Result is 0
Result = BitTest(11, 3)        ;* Result is 1
*  (bin) (dec) BitTest bit#  is:
*  1011    11           0     1
*  1011    11           1     1
*  1011    11           2     0
*  1011    11           3     1
```

BitXOr

```
Result = BitXor(6, 12)         ;* Result is 10
*  (bin) (dec) BitXOr (bin) (dec) gives (bin) (dec)
*   110     6         1100    12        1010    10
```

Byte-Oriented Functions

IBM InfoSphere DataStage provides four functions that can be used to manipulate internal strings at the byte level.

- **Byte** lets you build a string byte by byte.
- **ByteLen** returns the length of a string in bytes.
- **ByteType** determines the internal function of a particular byte.

- **ByteVal** determines the value of a particular byte in a string.

  Note: Use these functions with care: if you create an invalid string, it could produce unexpected results when processed by another function.

Byte Function

Returns a byte from an input numerical value.

Syntax

Byte (*expression*)

expression is a character value in the range 0 through 255.

Remarks

The **Byte** function can be used to build a string byte by byte, rather than character by character. If NLS is not enabled, the **Byte** function works like the **Char** function.

ByteLen Function

Returns the length of an internal string in bytes, rather than characters.

Syntax

ByteLen (*expression*)

expression is the string to be evaluated.

Remarks

If *expression* is an empty string, the result is 0. If *expression* is an SQL null, the result is a null.

ByteType Function

Returns the function of a particular byte within an internal character code.

Syntax

ByteType (*value*)

value is a byte value, 0 through 255, whose function is to be determined. If *value* is an SQL null, a null is returned.

Remarks

The result is returned as one of the following values:

Value	Meaning
0	The trailing byte of a multibyte character
1	A single-byte character
2	The lead byte of a two-byte character
3	The lead byte of a three-byte character

4 Reserved (lead byte of a four-byte character

5 A system delimiter

-1 The input value is not in the range 0 through 255

ByteVal Function

Returns the internal value for a specified byte in a string.

ByteVal

Syntax

```
ByteVal ( string  [, byte_number ]  )
```

string contains the byte to evaluate. An empty string or null value returns -1. A string that has fewer bytes than specified in *byte_number* returns -1.

byte_number is the number of the byte in *string* to evaluate. If omitted or less than 1, 1 is used.

Remarks

The result is returned as a value for the byte in the range 0 through 255.

Call Statement

Calls a subroutine. Not available in expressions.

Syntax

```
Call subroutine [ ( argument [ , argument ] ... ) ]
```

argument is a variable, expression, or constant that you want to pass to the subroutine. Multiple arguments must be separated by commas.

Remarks

Call transfers program control from the main program to a compiled external subroutine. Use a **Return** statement to return control to the main program.

The number of arguments specified in a **Call** statement must match the number of arguments specified in the **Subroutine** statement that identifies the subroutine.

Constants are passed by value; variables are passed by reference. If you want to pass variables by value, enclose them in parentheses.

Note: If you pass variables by value, any change to the variable in the subroutine does not affect the value of the variable in the main program. If you pass variables by reference, any change to the variable in the subroutine also affects the main program.

Example

This example shows how to call a before/after routine named MyRoutineB from within another routine called MyRoutineA:

```
Subroutine MyRoutineA(InputArg, ErrorCode)
   ErrorCode = 0         ;* set local error code
   * When calling a user-written routine that is held in the
   * DataStage repository, you must add a "DSU." Prefix.
   * Be careful to supply another variable for the called
   * routine's 2nd argument so as to keep separate from our
   * own.
   Call DSU.MyRoutineB("First argument", ErrorCodeB)
   If ErrorCodeB <> 0 Then
      ... ;* called routine failed - take action
   Endif
Return
```

Case Statement

Alters the sequence of execution in the program according to the value of an expression. Not available in expressions.

Syntax

Begin Case **Case** *expression* *statements* [**Case** *expression* *statements*] ...
End Case

expression is a value used to test the case. If *expression* is a null value, it is assumed to be false.

statements are the statements to execute if *expression* is true.

Remarks

Case statements can be repeated. If *expression* in the first **Case** statement is true, the following statements are executed. If *expression* is false, the program moves to the next **Case** statement. The process is repeated until an **End Case** statement is reached.

If more than one expression is true, only the first one is acted on. If no expression is true, none of the statements are executed.

To test if a variable contains a null value, use this syntax:

Case IsNull (*expression*)

To specify a default case to execute if all other expressions are false, use an expression containing the constant value 1.

Example

This example uses **Case** statements on the incoming argument to select the type of processing to perform within a routine:

```
Function MyTransform(Arg1)
   Begin Case
   Case Arg1 = 1
      Reply = "A"
   Case Arg1 = 2
      Reply = "B"

   Case Arg1 > 2 And Arg1 < 11
      Reply = "C"
   Case @True      ;* all other values
      Call DSTransformError("Bad arg":Arg1, "MyTransform"
      Reply = ""
   End Case
Return(Reply)
```

Cats Statement

Concatenates two strings.

Syntax

Cats (*string1*, *string2*)

string1, *string2* are the strings to be concatenated. If either string is a null value, a null value is returned.

Example

```
String1 = "ABC"
String2 = "1234"
Result = Cats(String1, String2)
* Result contains "ABC1234"
```

Change Function

Replaces one or more instances of a substring.

Syntax

Change (*string*, *substring*, *replacement* **[,***number* **[,***start***]])**

string is the string or expression in which you want to change substrings. If *string* evaluates to a null value, null is returned.

substring is the substring you want to replace. If it is empty, the value of *string* is returned (this is the only difference between **Change** and **Ereplace**).

replacement is the replacement substring. If *replacement* is an empty string, all occurrences of *substring* are removed.

number specifies the number of instances of *substring* to replace. To change all instances, use a value less than 1.

start specifies the first instance to replace. A value less than 1 defaults to 1.

Remarks

A null value for *string* returns a null value. If you use a null value for any other variable, a runtime error occurs.

Examples

The following example replaces all occurrences of one substring with another:

```
MyString = "AABBCCBBDDBB"NewString = Change(MyString, "BB",
→ "xxx") * The result is "AAxxxCCxxxDDxxx"
```

The following example replaces only the first two occurrences.

```
MyString = "AABBCCBBDDBB"NewString = Change (MyString, "BB",
→ "xxx", 2, 1)* The result is "AAxxxCCxxxDDBB"
```

The following example removes all occurrences of the substring:

```
MyString = "AABBCCBBDDBB"NewString = Change (MyString, "BB",
→ "")* The result is "AACCDD"
```

Char Function

Generates an ASCII character from its numeric code value.

Syntax

Char (*code*)

code is the ASCII code value of the character or an expression evaluating to the code.

Remarks

Be careful with null values. If *code* is a null value, null is returned. If *code* is 128, the returned value is CHAR(128), that is, the system variable @NULL.STR.

The **Char** function is the inverse of the **Seq** function.

Note: If NLS is enabled, values for code in the range 129 through 247 return Unicode values in the range x0081 through x00F7. These are multibyte characters equivalent to the same values in the ISO 8859 (Latin 1) character set. To generate the specific bytes with the values 129 through 247, use the **Byte** function.

Example

This example uses the **Char** function to return the character associated with the specified character code:

```
MyChar = Char(65)    ;* returns "A"
MyChar = Char(97)    ;* returns "a"
MyChar = Char(32)    ;* returns a space
MyChar = Char(544)
* returns a space (544 = 32 modulus 256)
```

Checksum Function

Returns a checksum value for a string.

Syntax

Checksum (*string*)

string is the string you want to add the checksum to. If *string* is a null value, null is returned.

Example

This example uses the **Checksum** function to return a number that is a cyclic redundancy code for the specified string:

```
MyString = "This is any arbitrary string value"
CheckValue = Checksum(MyString)  ;* returns 36235
```

CloseSeq Statement

Closes a file after sequential processing.

Syntax

CloseSeq *file.variable* [**On Error** *statements*]

file.variable specifies a file previously opened with an **OpenSeq** statement.

On Error *statements* specifies statements to execute if a fatal error occurs during processing of the **CloseSeq** statement.

Remarks

Each sequential file reference in a routine must be preceded by a separate **OpenSeq** statement for that file. **OpenSeq** sets an update record lock on the file. This prevents any other program from changing the file while you are processing it. **CloseSeq** resets this lock after processing the file. Multiple **OpenSeq** operations on the same file only generate one update record lock so you need only include one **CloseSeq** statement per file.

If a fatal error occurs, and no **On Error** clause was specified:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.

If the **On Error** clause is taken, the value returned by the **Status** function is the error number.

Col1 Function

Returns the character position preceding the substring specified in the most recently executed **Field** function.

Syntax

```
Col1 ( )
```

Remarks

The character position is returned as a number. The returned value is local to the routine executing the **Field** function. The value of **Col1** in the routine is initialized as 0.

Col1 returns a value of 0 if:

- No **Field** function was executed.
- The delimiter expression of the **Field** function is an empty string or the null value.
- The string is not found.

Examples

The **Field** function in the following example returns substring "CCC". **Col1** ( ) returns 8, the position of the delimiter (/) that precedes CCC.

```
* Extract third "/"-delimited field.
SubString = Field("AAA/BBB/CCC", "/" ,3)
Position = Col1()       ;* get position of delimiter
```

In the following example, the **Field** function returns a substring of two fields with the delimiter (.) that separates them: 4.5. **Col1** ( ) returns 6, the position of the delimiter that precedes 4.

```
* Get fourth and fifth "."-delimited fields.
SubString = Field("1.2.3.4.5.6", ".", 4, 2)
Position = Col1()        ;* get position of delimiter
```

Col2 Function

Returns the character position following the substring specified in the most recently executed **Field** function.

Syntax

Col2 ( )

Remarks

The character position is returned as a number. The returned value is local to the routine executing the **Field** function. The value of **Col2** in the routine is initialized as 0. When control is returned to the calling program, the saved value of **Col2** is restored.

Col2 returns a value of 0 if:

- No **Field** function was executed.
- The delimiter expression of the **Field** function is an empty string or the null value.
- The string is not found.

Examples

The **Field** function in the following example returns substring "CCC". **Col2** ( ) returns 12, the position that the delimiter (/) would have occupied following CCC if the end of the string had not been encountered.

```
* Extract third "/"-delimited field.
SubString = Field("AAA/BBB/CCC", "/" ,3)
Position = Col2()       ;* returns end of string in fact
```

In the following example, the **Field** function returns a substring of two fields with the delimiter (.) that separates them: 4.5. **Col2** ( ) returns 10, the position of the delimiter that follows 5.

```
* Get fourth and fifth "."-delimited fields.
SubString = Field("1.2.3.4.5.6", ".", 4, 2)
Position = Col2()      ;* get position of delimiter
```

In the next example, **Field** returns the whole string, because the delimiter (.) is not found. **Col2** ( ) returns 6, the position after the last character of the string.

```
* Attempts to first get first "."-delimited field,
* but fails.
SubString = Field("9*8*7", ".", 1)
Position = Col2()        ;* returns length of string + 1
```

In the next example, **Field** returns an empty string, because there is no tenth occurrence of the substring in the string. **Col2** ( ) returns 0 because the substring was not found.

```
* Attempts to first get tenth "."-delimited
* field, but fails.
SubString = Field("9*8*7*6*5*4", "*", 10)
Position = Col2               ;* returns 0
```

Common Statement

Defines a common storage area for variables. Not available in expressions.

Syntax

Common */name* / *variable* [,*variable*] ...

/name/ is the name identifying the common area and is significant to 31 characters.

variable is the name of a variable to store in the common area.

Remarks

Variables in the common area are accessible to all routines that have the */name/* common declared. (Use the **$Include** statement to define the common area in each routine.) Corresponding variables can have different names in different routines, but they must be defined in the same order. The **Common** statement must precede any reference to the variables it names.

Arrays can be dimensioned and named with a **Common** statement. They can be redimensioned later with a **Dimension** statement, but the **Common** statement must appear before the **Dimension** statement.

Example

This example shows two routines communicating via a common area named MyCommon, defined in a separate file in the DSU_BP subdirectory whose name is declared by a **$Include** statement:

The file DSU_BP \ MyCommon.H contains:

```
Common /MyCommon/               ComVar1,   ;* single variable
               ComVar2(10)   ;* array of 10 variables
```

The routines are defined as before/afters, as follows:

```
Subroutine MyRoutineA(InputArg, ErrorCode)
$Include MyCommon.H
   ErrorCode = 0
* Distribute fields of incoming argument into common * array:
   For n = 1 To 10
      ComVar2(n) = Field(InputArg, ",", n)
      If ComVar2(n) <> "" Then
         ComVar1 = n     ;* indicate highest one used
      End
   Next n
   Call DSU.MyRoutineB("another arg", ErrorCodeB)
* Etc.
...
Return
Subroutine MyRoutineB(InputArg, ErrorCode)
$Include MyCommon.H
   ErrorCode = 0
* Read the values out of the common array:
   For n = 1 To ComVar1
      MyVar = ComVar2(n)
      * Do something with it...
      ...
   Next n
Return
```

Compare Function

Compares two strings. If NLS is enabled, the result of this function depends on the current locale setting of the Collate convention.

Syntax

Compare (*string1*, *string2* [, *justification*])

string1, *string2* are the strings to be compared.

justification is either L for left-justified comparison or R for right-justified comparison. If you do not specify L or R, L is the default. Any other value causes a runtime warning, and 0 is returned.

Remarks

The result of the comparison is returned as one of the following values:

-1 *string1* is less than *string2*.

0 *string1* equals *string2* or the justification expression is not valid.

1 *string1* is greater than *string2*.

Use a right-justified comparison for numeric strings; use a left-justified comparison for text strings. For mixed strings, take care. For example, a right-justified comparison of the strings AB100 and AB99 indicates that AB100 is greater than AB99 since 100 is greater than 99. But a right-justified comparison of the strings AC99 and AB100 indicates that AC99 is greater since C is greater than B.

Example

In the following example, the strings AB99 and AB100 are compared with the right-justified option, in which "AB100" is greater than "AB99":

```
On Compare("AB99", "AB100", "R") + 2 GoSub
    LessThan,
    EqualTo
    GreaterThan
```

Convert Function

Replaces every instance of specified characters in a string with substitute characters.

Syntax

Convert (*list*, *new.list*, *string*)

list is a list of characters to replace. If *list* is a null value it generates a runtime error.

new.list is a corresponding list of substitute characters. If *new.list* is a null value, it generates a runtime error.

string is an expression that evaluates to the string, or a variable containing the string. If *string* is a null value, null is returned.

Remarks

The two lists of characters correspond. The first character of *new.list* replaces all instances of the first character of list, the second replaces the second, and so on. If the two lists do not contain the same number of characters:

- Any characters in *list* with no corresponding characters in *new.list* are deleted from the result.
- Any surplus characters in *new.list* are ignored.

Example

This is an example of **Convert** used as a function:

```
MyString ="NOW IS THE TIME"
ConvStr = Convert("TI", "XY", MyString)
* all T => X, I => Y
* At this point ConvStr is: NOW YS XHE XYME
ConvStr = Convert("XY", "Z", ConvStr)
* all X => Z, Y => ""
* At this point ConvStr is: NOW S ZHE ZME
```

Convert Statement

Replaces every instance of specified characters in a string with substitute characters. Not available in expressions.

Syntax

Convert *list* **To** *new.list* **In** *string*

list is a list of characters to replace. If *list* is a null value, it generates a runtime error.

new.list is a corresponding list of substitute characters. If *new.list* is a null value, it generates a runtime error.

string is an expression that evaluates to the string, or a variable containing the string. If *string* is a null value, null is returned.

Remarks

The two lists of characters correspond. The first character of *new.list* replaces all instances of the first character of list, the second replaces the second, and so on. If the two lists do not contain the same number of characters:

- Any characters in *list* with no corresponding characters in *new.list* are deleted from the result.
- Any surplus characters in *new.list* are ignored.

Example

This example shows **Convert** used as a statement, converting the string in place:

```
MyString ="NOW IS THE TIME"
Convert "TI" To "XY" In MyString
* all T => X, I => Y
* At this point MyString is: NOW YS XHE XYME
Convert "XY" To "Z" In MyString
* all X => Z, Y => ""
* At this point MyString is: NOW S ZHE ZME
```

Count Function

Counts the number of times a substring occurs in a string.

Syntax

Count (*string*, *substring*)

string is the string you want to search. If *string* is a null value, null is returned.

substring is the substring you want to count. It can be a character string, a constant, or a variable. If *substring* does not appear in *string*, 0 is returned. If *substring* is an empty string, the number of characters in *string* is returned. If *substring* is a null value, a runtime error results.

Remarks

When one complete substring is counted, **Count** moves on to the next character and starts again. For example, the following statement counts only two instances of substring tt and returns 2 to variable c:

```
c = Count ('tttt', 'tt')
```

Example

```
* The next line returns the number of "A"s
* in the string (3).
MyCount = Count("ABCAGHDALL", "A")
* The next line returns 2 since overlapping substrings
* are not counted.
MyCount = Count ("TTTT", "TT")
```

CRC32 Function

Returns a 32-bit cyclic redundancy check value for a string.

Syntax

CRC32 (*string*)

string is the string you want to add the CRC value to. If *string* is a null value, null is returned.

Example

This example uses the **CRC** function to return a number that is a cyclic redundancy code for the specified string:

```
MyString = "This is any arbitrary string value"
CheckValue = CRC32(MyString)  ;* returns 36235
```

Date Function

Returns a date in its internal system format.

Syntax

Date ( )

Remarks

IBM InfoSphere DataStage stores dates as the number of days before or after day 0, using 31 December 1967 as day 0. For example:

This date...
Is stored as...

December 10, 1967
-21

November 15, 1967
-46

December 31, 1967
0

February 15, 1968
46

January 1, 1985
6575

Use the internal date whenever you need to perform output conversions.

Example

This example shows how to turn the current date in internal form into a string representing the next day:

```
Tomorrow = Oconv(Date() + 1, "D4/YMD") ;* "1997/5/24"
```

DCount Function

Counts delimited fields in a string.

Syntax

```
DCount (string, delimiter)
```

string is the string to be searched. If *string* is an empty string, 0 is returned. If *string* is a null value, null is returned.

delimiter is one or more characters delimiting the fields to be counted. If *delimiter* is an empty string, the number of characters in string + 1 is returned. If *delimiter* is a null value, a runtime error occurs. Two consecutive delimiters in string are counted as one field.

Remarks

DCount differs from **Count** in that it returns the number of values separated by delimiters rather than the number of occurrences of a character string.

Example

```
* The next line returns the number of substrings
* delimited by "A"s in the string (4)
MyCount = DCount("ABCAGHDALL", "A")
* The next line returns 3 since overlapping substrings
* are not counted.
MyCount = DCount ("TTTT", "TT")
```

Deffun Statement

Defines a user-written function.

Syntax

```
Deffun function [ ( [Mat] argument [, [Mat] argument ...] ) ]
[Calling call.name]
```

function is the name of the function to be defined.

argument is an argument to pass to the function. You can supply up to 254 arguments. To pass an array, precede the array name with **Mat**.

Calling *call.name* specifies the name used to call the function. If you do not specify a name, the function is called using *function*.

Remarks

You must declare a user-written function before you can use it in a program. You can define a user-written function only once in a program. Defining the function twice causes a fatal error.

Example

This example shows how to define a transform function named MyFunctionB so that it can be called from within another transform function named MyFunctionA:

```
Function MyFunctionA(Arg1)
* When referencing a user-written function that is held in the
* DataStage repository, you must declare it as a function with
* the correct number of arguments, and add a "DSU." prefix.
Deffun MyFunctionB(A) Calling "DSU.MyFunctionB"
```

Dimension Statement

Defines the dimensions of one or more arrays. Not available in expressions.

Syntax

Dimension *matrix* (*rows*, *columns*) [,*matrix* (*rows*, *columns*)] ...
Dimension *vector* (*max*) [, *vector* (*max*)] ...

matrix is a two-dimensional array to be dimensioned.

rows is the maximum number of rows in the array.

columns is the maximum number of columns in the array.

vector is a one-dimensional array to be dimensioned.

max is the maximum number of elements in the array.

Remarks

Arrays can be redimensioned at run time. You can change an array from one-dimensional to two-dimensional and vice versa.

The values of array elements are affected by redimensioning as follows:

- Common elements with the same *row/column* address in both arrays are preserved.
- New elements that had no *row/column* address in the original array are initialized as unassigned.
- Redundant elements that can no longer be referenced in the new array are lost, and the memory space is returned to the operating system.

If there is not enough memory for the array, the **Dimension** statement fails and a following **InMat** function returns 1.

To assign values to the elements of the array, use the **Mat** statement and assignment statements.

Example

This example illustrates how a matrix can be dimensioned dynamically at run time based on incoming argument values:

```
Subroutine MyRoutine(InputArg, ErrorCode)
   ErrorCode = 0
* InputArg is 2 comma-separated fields, being the dimensions.
   Rows = Field(InputArg, ",", 1)
   Cols = Field(InputArg ",", 2)
   Dimension MyMatrix(Rows, Cols)
   If InMat = 1 Then
   * Failed to get space for matrix - exit with error status.
      Call DSLogWarn("Could not dimension matrix","MyRoutine")
      ErrorCode = -1
   Else
   * Carry on.
      ...
   End
```

Div Function

Divides one number by another.

Syntax

Div (*dividend*, *divisor*)

dividend is the number to be divided. If *dividend* is a null value, null is returned.

divisor is the number to divide by. *divisor* cannot be 0. If *divisor* is a null value, null is returned.

Remarks

Use the **Mod** function to determine any remainder.

Examples

The following examples show use of the **Div** function:

```
Quotient = Div(100, 25)              ;* result is 4
Quotient = Div(100, 30)              ;* result is 3
```

DownCase Function

Converts uppercase letters in a string to lowercase. If NLS is enabled, the result of this function depends on the current locale setting of the Ctype convention.

Syntax

DownCase (*string*)

string is a string or expression to change to lowercase. If *string* is a null value, null is returned.

Example

This is an example of the **DownCase** function:

```
MixedCase = "ABC123abc"
LowerCase = DownCase(MyString) ;* result is "abc123abc"
```

DQuote Function

Encloses a string in double quotation marks.

Syntax

```
DQuote (string)
```

string is the string to be quoted. If *string* is a null value, null is returned.

Remarks

To enclose a string in single quotation marks, use the **SQuote** function.

Example

This is an example of the **DQuote** function adding double quotation marks (") to the start and end of a string:

```
ProductNo = 12345
QuotedStr = DQuote(ProductNo : "A")
* result is "12345A"
```

DSAttachJob

Attaches to a job in order to run it in job control sequence. A handle is returned which is used for addressing the job. There can only be one handle open for a particular job at any one time.

Syntax

```
JobHandle = DSAttachJob (JobName, ErrorMode)
```

JobHandle is the name of a variable to hold the return value which is subsequently used by any other function or routine when referring to the job. Do not assume that this value is an integer.

JobName is a string giving the name of the job to be attached to.

ErrorMode is a value specifying how other routines using the handle should report errors. It is one of:

- DSJ.ERRFATAL Log a fatal message and abort the controlling job (default).
- DSJ.ERRWARNING Log a warning message but carry on.
- DSJ.ERRNONE No message logged - caller takes full responsibility (failure of DSAttachJob itself will be logged, however).

Remarks

A job cannot attach to itself.

The *JobName* parameter can specify either an exact version of the job in the form *job*%Rel*n.n.n*, or the latest version of the job in the form *job*. If a controlling job is itself released, you will get the latest released version of *job*. If the controlling job is a development version, you will get the latest development version of *job*.

Example

This is an example of attaching to Release 11 of the job Qsales:

```
Qsales_handle = DSAttachJob ("Qsales%Rel1",
→ DSJ.ERRWARN)
```

DSCheckRoutine

Checks if a BASIC routine is cataloged, either in the VOC as a callable item, or in the catalog space.

Syntax

```
Found = DSCheckRoutine(RoutineName)
```

RoutineName is the name of BASIC routine to check.

Found Boolean. @False if *RoutineName* not findable, else @True.

Example

```
rtn$ok = DSCheckRoutine("DSU.DSSendMail")
If(NOT(rtn$ok)) Then
   * error handling here
End.
```

DSDetachJob

Gives back a *JobHandle* acquired by DSAttachJob if no further control of a job is required (allowing another job to become its controller). It is not necessary to call this function, otherwise any attached jobs will always be detached automatically when the controlling job finishes.

Syntax

```
ErrCode = DSDetachJob (JobHandle)
```

JobHandle is the handle for the job as derived from DSAttachJob.

ErrCode is 0 if DSStopJob is successful, otherwise it might be the following:

- DSJE.BADHANDLE Invalid *JobHandle*.

The only possible error is an attempt to close DSJ.ME. Otherwise, the call always succeeds.

Example

The following command detaches the handle for the job qsales:

```
Deterr = DSDetachJob (qsales_handle)
```

DSExecute

Executes a DOS, UNIX, or engine command from a before/after subroutine.

Syntax

```
Call DSExecute (ShellType, Command, Output, SystemReturnCode)
```

ShellType (input) specifies the type of command that you want to execute and is NT, UNIX, or UV (for engine).

Command (input) is the command to execute. *Command* should not prompt for input when it is executed.

Output (output) is any output from the command. Each line of output is separated by a field mark, @FM. Output is added to the job log file as an information message.

SystemReturnCode (output) is a code indicating the success of the command. A value of 0 means the command executed successfully. A value of 1 (for a DOS or UNIX command) indicates that the command was not found. Any other value is a specific exit code from the command.

Remarks

Do not use DSExecute from a transform; the overhead of running a command for each row processed by a stage will degrade performance of the job.

DSGetCustInfo

Obtains information reported at the end of execution of certain parallel stages. The information collected, and available to be interrogated, is specified at design time. For example, transformer stage information is specified in the Triggers tab of the Transformer stage Properties dialog box.

Syntax

```
Result = DSGetCustInfo (JobHandle, StageName, CustInfoName, InfoType)
```

JobHandle is the handle for the job as derived from DSAttachJob, or it might be DSJ.ME to refer to the current job.

StageName is the name of the stage to be interrogated. It might also be DSJ.ME to refer to the current stage if necessary.

CustInfoName is the name of the variable to be interrogated.

InfoType specifies the information required and can be one of:

DSJ.CUSTINFOVALUE

DSJ.CUSTINFODESC

Result depends on the specified *InfoType*, as follows:

- DSJ.CUSTINFOVALUE String - the value of the specified custinfo item.
- DSJ.CUSTINFODESC String - description of the specified custinfo item.

Result might also return an error condition as follows:

- DSJE.BADHANDLE *JobHandle* was invalid.
- DSJE.BADTYPE *InfoType* was unrecognized.
- DSJE.NOTINSTAGE *StageName* was DSJ.ME and the caller is not running within a stage.
- DSJE.BADSTAGE *StageName* does not refer to a known stage in the job.
- DSJE.BADCUSTINFO *CustInfoName* does not refer to a known custinfo item.

DSGetJobInfo

Provides a method of obtaining information about a job, which can be used generally as well as for job control. It can refer to the current job or a controlled job, depending on the value of *JobHandle*.

Syntax

```
Result = DSGetJobInfo (JobHandle, InfoType)
```

JobHandle is the handle for the job as derived from DSAttachJob, or it might be DSJ.ME to refer to the current job.

InfoType specifies the information required and can be one of:

DSJ.JOBSTATUS

DSJ.JOBNAME

DSJ.JOBCONTROLLER

DSJ.JOBSTARTTIMESTAMP

DSJ.JOBWAVENO

DSJ.PARAMLIST

DSJ.STAGELIST

DSJ.USERSTATUS

DSJ.JOBCONTROL

DSJ.JOBPID

DSJ.JPBLASTTIMESTAMP

DSJ.JOBINVOCATIONS

DSJ.JOBINTERIMSTATUS

DSJ.JOBINVOCATIONID

DSJ.JOBDESC

DSJ.JOBFULLDESC

DSJ.STAGELIST2

DSJ.JOBELAPSED

DSJ.JOBEOTCOUNT

DSJ.JOBEOTTIMESTAMP

DSJ.JOBRTISERVICE

DSJ.JOBMULTIINVOKABLE

DSJ.JOBFULLSTAGELIST

Result depends on the specified *InfoType*, as follows:

- DSJ.JOBSTATUS *Integer*. Current status of job overall. Possible statuses that can be returned are currently divided into two categories:
  Firstly, a job that is in progress is identified by:
  DSJS.RESET Job finished a reset run.
  DSJS.RUNFAILED Job finished a normal run with a fatal error.
  DSJS.RUNNING Job running - this is the only status that means the job is actually running.
  Secondly, jobs that are not running might have the following statuses:
  DSJS.RUNOK Job finished a normal run with no warnings.
  DSJS.RUNWARN Job finished a normal run with warnings.

DSJS.STOPPED Job was stopped by operator intervention (can't tell run type).

DSJS.VALFAILED Job failed a validation run.

DSJS.VALOK Job finished a validation run with no warnings.

DSJS.VALWARN Job finished a validation run with warnings.

- DSJ.JOBNAME *String*. Actual name of the job referenced by the job handle.
- DSJ.JOBCONTROLLER *String*. Name of the job controlling the job referenced by the job handle. Note that this might be several job names separated by periods if the job is controlled by a job which is itself controlled.
- DSJ.JOBSTARTTIMESTAMP *String*. Date and time when the job started on the engine in the form *YYYY-MM-DD hh:nn:ss*.
- DSJ.JOBWAVENO *Integer*. Wave number of last or current run.
- DSJ.PARAMLIST. Returns a comma-separated list of parameter names.
- DSJ.STAGELIST. Returns a comma-separated list of active stage names.
- DSJ.USERSTATUS *String*. Whatever the job's last call of DSSetUserStatus last recorded, else the empty string.
- DSJ.JOBCONTROL *Integer*. Current job control status, that is, whether a stop request has been issued for the job.
- DSJ. JOBPID *Integer*. Job process id.
- DSJ.JOBLASTTIMESTAMP *String*. Date and time when the job last finished a run on the engine in the form *YYYY-MM-DD HH:NN:SS*.
- DSJ.JOBINVOCATIONS. Returns a comma-separated list of Invocation IDs.
- DSJ.JOBINTERIMSTATUS. Returns the status of a job after it has run all stages and controlled jobs, but before it has attempted to run an after-job subroutine. (Designed to be used by an after-job subroutine to get the status of the current job).
- DSJ.JOBINVOCATIONID. Returns the invocation ID of the specified job (used in the DSJobInvocationId macro in a job design to access the invocation ID by which the job is invoked).
- DSJ.STAGELIST2. Returns a comma-separated list of passive stage names.
- DSJ.JOBELAPSED *String*. The elapsed time of the job in seconds.
- DSJ.JOBDESC *string*. The Job Description specified in the Job Properties dialog box.
- DSJ.JOBFULLDESSC *string*. The Full Description specified in the Job Properties dialog box.
- DSJ.JOBRTISERVICE *integer*. Set to true if this is a Web service job.
- DSJ.JOBMULTIINVOKABLE *integer*. Set to true if this job supports multiple invocations
- DSJ.JOBEOTCOUNT *integer*. Count of EndOfTransmission blocks processed by this job so far.
- DSJ.JOBEOTTIMESTAMP *timestamp*. Date/time of the last EndOfTransmission block processed by this job.
- DSJ.FULLSTAGELIST. Returns a comma-separated list of all stage names.

Result might also return error conditions as follows:

DSJE.BADHANDLE *JobHandle* was invalid.

DSJE.BADTYPE *InfoType* was unrecognized.

Remarks

When referring to a controlled job, DSGetJobInfo can be used either before or after a DSRunJob has been issued. Any status returned following a successful call to DSRunJob is guaranteed to relate to that run of the job.

Examples

The following command requests the job status of the job qsales:

```
q_status = DSGetJobInfo(qsales_handle, DSJ.JOBSTATUS)
```

The following command requests the actual name of the current job:

```
whatname = DSGetJobInfo (DSJ.ME, DSJ.JOBNAME)
```

DSGetJobMetaBag

Returns a dynamic array containing the MetaBag properties associated with the named job.

Syntax

```
Result  =  DSGetJobMetaBag(JobName, Owner)
or
Call DSGetJobMetaBag(Result, JobName, Owner)
```

JobName is the name of the job in the current project for which information is required. If *JobName* does not exist in the current project *Result* will be set to an empty string.

Owner is an owner name whose metabag properties are to be returned. If *Owner* is not a valid owner within the current job, *Result* will be set to an empty string. If *Owner* is an empty string, a field mark delimited string of metabag property owners within the current job will be returned in *Result*.

Result returns a dynamic array of metabag property sets, as follows:

RESULT<1> = MetaPropertyName01 @VM MetaPropertyValue01

RESULT<..> = MetaPropertyName.. @VM MetaPropertyValue..

RESULT<*N*>= MetaPropertyName*N* @VM MetaPropertyValue*N*

Example

The following returns the metabag properties for owner *mbowner* in the job "testjob":

```
linksmdata = DSGetJobMetaBag (testjob, mbowner)
```

DSGetLinkInfo

Provides a method of obtaining information about a link on an active stage, which can be used generally as well as for job control. This routine might reference either a controlled job or the current job, depending on the value of *JobHandle*.

Syntax

```
Result = DSGetLinkInfo (JobHandle, StageName, LinkName, InfoType)
```

JobHandle is the handle for the job as derived from DSAttachJob, or it can be DSJ.ME to refer to the current job.

StageName is the name of the active stage to be interrogated. might also be DSJ.ME to refer to the current stage if necessary.

LinkName is the name of a link (input or output) attached to the stage. might also be DSJ.ME to refer to current link (for example, when used in a Transformer expression or transform function called from link code).

InfoType specifies the information required and can be one of:

DSJ.LINKLASTERR

DSJ.LINKNAME

DSJ.LINKROWCOUNT

DSJ.LINKSQLSTATE

DSJ.LINKDBMSCODE

DSJ.LINKDESC

DSJ.LINKSTAGE

DSJ.INSTROWCOUNT

DSJ.LINKEOTROWCOUNT

Result depends on the specified *InfoType*, as follows:

- DSJ.LINKLASTERR String - last error message (if any) reported from the link in question.
- DSJ.LINKNAME String - returns the name of the link, most useful when used with *JobHandle* = DSJ.ME and *StageName* = DSJ.ME and *LinkName* = DSJ.ME to discover your own name.
- DSJ.LINKROWCOUNT Integer - number of rows that have passed down a link so far.
- DSJ.LINKSQLSTATE - the SQL state for the last error occurring on this link.
- DSJ.LINKDBMSCODE - the DBMS code for the last error occurring on this link.
- DSJ.LINKDESC - description of the link.
- DSJ.LINKSTAGE - name of the stage at the other end of the link.
- DSJ.INSTROWCOUNT - comma-separated list of row counts, one per instance (parallel jobs)
- DSJ.LINKEOTROWCOUNT - row count since last EndOfTransmission block.

Result might also return error conditions as follows:

- DSJE.BADHANDLE *JobHandle* was invalid.
- DSJE.BADTYPE *InfoType* was unrecognized.
- DSJE.BADSTAGE *StageName* does not refer to a known stage in the job.
- DSJE.NOTINSTAGE *StageName* was DSJ.ME and the caller is not running within a stage.
- DSJE.BADLINK *LinkName* does not refer to a known link for the stage in question.

Remarks

When referring to a controlled job, DSGetLinkInfo can be used either before or after a DSRunJob has been issued. Any status returned following a successful call to DSRunJob is guaranteed to relate to that run of the job.

Example

The following command requests the number of rows that have passed down the order_feed link in the loader stage of the job qsales:

```
link_status = DSGetLinkInfo(qsales_handle, "loader",
→ "order_feed", DSJ.LINKROWCOUNT)
```

DSGetLinkMetaData

Returns a dynamic array containing the column metadata of the specified link.

Syntax

```
Result = DSGetLinkMetaData(JobName, StageName, LinkName)
         or
Call DSGetLinkMetaData(Result, JobName, StageName, LinkName)
```

JobName is the name of the job in the current project for which information is required. If the JobName does not exist in the current project then the function will return an empty string.

StageName is the name of the stage in the specified job containing the link for which information is required. If the StageName does not exist in the specified job then the function will return an empty string.

LinkName is the name of the link in the specified job for which information is required. If the LinkName does not exist in the specified job then the function will return an empty string.

Result returns a dynamic array of nine fields, each field will contain N values where N is the number of columns on the link.

Result<1,1...N> is the column name

Result<2,1...N> is 1 for primary key columns otherwise 0

Result<3,1...N> is the column SQL type. See ODBC.H.

Result<4,1...N> is the column precision

Result<5,1...N> is the column scale

Result<6,1...N> is the column display width

Result<7,1...N> is 1 for nullable columns otherwise 0

Result<8,1...N> is the column descriptions

Result<9,1...N> is the column derivation

Example

The following returns the metadata of the link ilink1 on the stage seqstage in the job testjob:

```
linksmdata = DSGetLinkMetaData (testjob, seqstage, ilink1)
```

DSGetLogEntry

Reads the full event details given in *EventId*.

Syntax

```
EventDetail = DSGetLogEntry (JobHandle, EventId)
```

JobHandle is the handle for the job as derived from DSAttachJob.

EventId is an integer that identifies the specific log event for which details are required. This is obtained using the DSGetNewestLogId function.

EventDetail is a string containing substrings separated by \. The substrings are as follows:

Substring1 Timestamp in form *YYYY-MM-DD HH:NN:SS*

Substring2 User information

Substring3 EventType - see DSGetNewestLogId

Substring4 - *n* Event message

If an error occurs, the error is reported by one of the following negative integer result codes:

- DSJE.BADHANDLE Invalid *JobHandle*.
- DSJE.BADVALUE Error accessing *EventId*.

Example

The following commands first get the EventID for the required log event and then reads full event details of the log event identified by LatestLogid into the string LatestEventString:

```
latestlogid =
→ DSGetNewestLogId(qsales_handle,DSJ.LOGANY)
LatestEventString =
→ DSGetLogEntry(qsales_handle,latestlogid)
```

DSGetLogEventIds

Returns a list of log event IDs for a given run of a job invocation.

Syntax

```
IdList = DSGetLogEventIds (JobHandle, RunNumber, EventTypeFilter)
```

JobHandle is the handle for the job as derived from DSAttachJob.

RunNumber identifies the job invocation run for which event IDs are returned. Usually a zero value requests IDs for the most recent run of the job invocation. To retrieve details for earlier runs, supply negative values, such as -1 for details about the run before the most recent, -2 for details about the run before that, and so forth. Where explicit run numbers are known, you can retrieve details by supplying the run number as a positive value.

EventTypeFilter restricts the types of event log entry for which IDs are returned. By default, IDs for all log entries are returned. Include characters in the filter string to restrict entries as follows:

I Informational

W Warning

F Fatal

S Start or End events

B Batch or Control events

R Purge or reset events

J Reject events

IdList is returned as a list of positive integers that identify the required log events. In the case of an error, *IdList* can also be returned as a negative integer, in which case it contains one of these error codes:

DSJE.BADHANDLE
Invalid *JobHandle*.

DSJE.BADTYPE
Invalid *EventTypeFilter*.

DSJE.BADVALUE
Invalid *RunNumber*.

Remarks

To use this method, the program needs to have previously acquired a job handle by calling DSAttachJob.

The run number for a job invocation is reset when the job is compiled, thus it is not possible to use this method to retrieve job event IDs for runs that occurred prior to the most recent job compilation.

DSGetLogSummary

Returns a list of short log event details. The details returned are determined by the setting of some filters. (Care should be taken with the setting of the filters, otherwise a large amount of information can be returned.)

Syntax

```
SummaryArray = DSGetLogSummary (JobHandle, EventType, StartTime, EndTime, MaxNumber)
```

JobHandle is the handle for the job as derived from *DSAttachJob*.

EventType is the type of event logged and is one of:
- DSJ.LOGINFO Information message
- DSJ.LOGWARNING Warning message
- DSJ.LOGFATAL Fatal error
- DSJ.LOGREJECT Reject link was active
- DSJ.LOGSTARTED Job started
- DSJ.LOGRESET Log was reset
- DSJ.LOGANY Any category (the default)

StartTime is a string in the form *YYYY-MM-DD HH:NN:SS* or *YYYY-MM-DD*.

EndTime is a string in the form *YYYY-MM-DD HH:NN:SS* or *YYYY-MM-DD*.

MaxNumber is an integer that restricts the number of events to return. 0 means no restriction. Use this setting with caution.

SummaryArray is a dynamic array of fields separated by @FM. Each field comprises a number of substrings separated by \, where each field represents a separate event, with the substrings as follows:

Substring1 *EventId* as per DSGetLogEntry

Substring2 Timestamp in form *YYYY-MM-DD HH:NN:SS*

Substring3 *EventType* - see DSGetNewestLogId

Substring4 - *n* Event message

If an error occurs, the error is reported by one of the following negative integer result codes:

- DSJE.BADHANDLE Invalid *JobHandle*.
- DSJE.BADTYPE Invalid *EventType*.
- DSJE.BADTIME Invalid *StartTime* or *EndTime*.
- DSJE.BADVALUE Invalid *MaxNumber*.

Example

The following command produces an array of reject link active events recorded for the qsales job between 18th August 1998, and 18th September 1998, up to a maximum of MAXREJ entries:

```
RejEntries = DSGetLogSummary (qsales_handle,
→ DSJ.LOGREJECT, "1998-08-18 00:00:00", "1998-09-18
→ 00:00:00", MAXREJ)
```

DSGetNewestLogId

Gets the ID of the most recent log event in a particular category, or in any category.

Syntax

```
EventId = DSGetNewestLogId (JobHandle, EventType)
```

JobHandle is the handle for the job as derived from DSAttachJob.

EventType is the type of event logged and is one of:

- DSJ.LOGINFO Information message
- DSJ.LOGWARNING Warning message
- DSJ.LOGFATAL Fatal error
- DSJ.LOGREJECT Reject link was active
- DSJ.LOGSTARTED Job started
- DSJ.LOGRESET Log was reset
- DSJ.LOGANY Any category (the default)

EventId is a positive integer that identifies the specific log event. In the case of an error, *EventId* can also be returned as a negative integer, in which case it contains an error code as follows:

- DSJE.BADHANDLE Invalid *JobHandle*.
- DSJE.BADTYPE Invalid *EventType*.

Example

The following command obtains an ID for the most recent warning message in the log for the qsales job:

```
Warnid = DSGetNewestLogId (qsales_handle,
→ DSJ.LOGWARNING)
```

DSGetParamInfo

Provides a method of obtaining information about a parameter, which can be used generally as well as for job control. This routine might reference either a controlled job or the current job, depending on the value of *JobHandle*.

Syntax

```
Result = DSGetParamInfo (JobHandle, ParamName, InfoType)
```

JobHandle is the handle for the job as derived from DSAttachJob, or it might be DSJ.ME to refer to the current job.

ParamName is the name of the parameter to be interrogated.

InfoType specifies the information required and might be one of:

DSJ.PARAMDEFAULT

DSJ.PARAMHELPTEXT

DSJ.PARAMPROMPT

DSJ.PARAMTYPE

DSJ.PARAMVALUE

DSJ.PARAMDES.DEFAULT

DSJ.PARAMLISTVALUES

DSJ.PARAMDES.LISTVALUES

DSJ.PARAMPROMPT.AT.RUN

Result depends on the specified *InfoType*, as follows:

- DSJ.PARAMDEFAULT String - Current default value for the parameter in question. See also DSJ.PARAMDES.DEFAULT.
- DSJ.PARAMHELPTEXT String - Help text (if any) for the parameter in question.
- DSJ.PARAMPROMPT String - Prompt (if any) for the parameter in question.
- DSJ.PARAMTYPE Integer - Describes the type of validation test that should be performed on any value being set for this parameter. Is one of:
  DSJ.PARAMTYPE.STRING
  DSJ.PARAMTYPE.ENCRYPTED
  DSJ.PARAMTYPE.INTEGER
  DSJ.PARAMTYPE.FLOAT (the parameter might contain periods and E)
  DSJ.PARAMTYPE.PATHNAME
  DSJ.PARAMTYPE.LIST (should be a string of Tab-separated strings)
  DSJ.PARAMTYPE.DATE (should be a string in form *YYYY-MM-DD*)
  DSJ.PARAMTYPE.TIME (should be a string in form *HH:MM*)
- DSJ.PARAMVALUE String - Current value of the parameter for the running job or the last job run if the job is finished.

- DSJ.PARAMDES.DEFAULT String - Original default value of the parameter - might differ from DSJ.PARAMDEFAULT if the latter has been changed by an administrator since the job was installed.
- DSJ.PARAMLISTVALUES String - Tab-separated list of allowed values for the parameter. See also DSJ.PARAMDES.LISTVALUES.
- DSJ.PARAMDES.LISTVALUES String - Original Tab-separated list of allowed values for the parameter - might differ from DSJ.PARAMLISTVALUES if the latter has been changed by an administrator since the job was installed.
- DSJ.PROMPT.AT.RUN String - 1 means the parameter is to be prompted for when the job is run; anything else means it is not (DSJ.PARAMDEFAULT String to be used directly).

Result might also return error conditions as follows:

- DSJE.BADHANDLE *JobHandle* was invalid.
- DSJE.BADPARAM *ParamName* is not a parameter name in the job.
- DSJE.BADTYPE *InfoType* was unrecognized.

Remarks

When referring to a controlled job, DSGetParamInfo can be used either before or after a DSRunJob has been issued. Any status returned following a successful call to DSRunJob is guaranteed to relate to that run of the job.

Example

The following command requests the default value of the quarter parameter for the qsales job:

```
Qs_quarter = DSGetparamInfo(qsales_handle, "quarter",
→ DSJ.PARAMDEFAULT)
```

DSGetProjectInfo

Provides a method of obtaining information about the current project.

Syntax

```
Result = DSGetProjectInfo (InfoType)
```

InfoType specifies the information required and can be one of:

DSJ.JOBLIST

DSJ.PROJECTNAME

DSJ.HOSTNAME

Result depends on the specified *InfoType*, as follows:

- DSJ.JOBLIST String - comma-separated list of names of all jobs known to the project (whether the jobs are currently attached or not).
- DSJ.PROJECTNAME String - name of the current project.
- DSJ.HOSTNAME String - the host name of the engine holding the current project.

Result might also return an error condition as follows:

- DSJE.BADTYPE *InfoType* was unrecognized.

DSGetStageInfo

Provides a method of obtaining information about a stage, which can be used generally as well as for job control. It can refer to the current job, or a controlled job, depending on the value of *JobHandle*.

Syntax

```
Result = DSGetStageInfo (JobHandle, StageName, InfoType)
```

JobHandle is the handle for the job as derived from DSAttachJob, or it might be DSJ.ME to refer to the current job.

StageName is the name of the stage to be interrogated. It might also be DSJ.ME to refer to the current stage if necessary.

InfoType specifies the information required and might be one of:

DSJ.LINKLIST

DSJ.STAGELASTERR

DSJ.STAGENAME

DSJ.STAGETYPE

DSJ.STAGEINROWNUM

DSJ.VARLIST

DSJ.STAGESTARTTIMESTAMP

DSJ.STAGEENDTIMESTAMP

DSJ.STAGEDESC

DSJ.STAGEINST

DSJ.STAGECPU

DSJ.LINKTYPES

DSJ.STAGEELAPSED

DSJ.STAGEPID

DSJ.STAGESTATUS

DSJ.STAGEEOTCOUNT

DSJ.STAGEEOTTIMESTAMP

DSJ.CUSTINFOLIST

DSJ.STAGEEOTSTART

Result depends on the specified *InfoType*, as follows:

- DSJ.LINKLIST - comma-separated list of link names in the stage.
- DSJ.STAGELASTERR String - last error message (if any) reported from any link of the stage in question.
- DSJ.STAGENAME String - most useful when used with *JobHandle* = DSJ.ME and *StageName* = DSJ.ME to discover your own name.
- DSJ.STAGETYPE String - the stage type name (for example, "Transformer", "BeforeJob").
- DSJ. STAGEINROWNUM Integer - the primary link's input row number.
- DSJ.VARLIST - comma-separated list of stage variable names.
- DSJ.STAGESTARTTIMESTAMP - date/time that stage started executing in the form *YYY-MM-DD HH:NN:SS.*
- DSJ.STAGEENDTIMESTAMP - date/time that stage finished executing in the form *YYY-MM-DD HH:NN:SS.*
- DSJ.STAGEDESC - stage description.
- DSJ.STAGEINST - comma-separated list of instance ids (parallel jobs).
- DSJ.STAGECPU - integer percentage of CPU used.
- DSJ.LINKTYPES - comma-separated list of link types.
- DSJ.STAGEELAPSED - elapsed time in seconds.
- DSJ.STAGEPID - comma-separated list of process ids.
- DSJ.STAGESTATUS - stage status.
- DSJ.STAGEEOTCOUNT - Count of EndOfTransmission blocks processed by this stage so far.
- DSJ.STAGEEOTTIMESTAMP - Data/time of last EndOfTransmission block received by this stage.
- DSJ.CUSTINFOLIST - custom information generated by stages (parallel jobs).
- DSJ.STAGEEOTSTART - row count at start of current EndOfTransmission block.

Result might also return error conditions as follows:
- DSJE.BADHANDLE *JobHandle* was invalid.
- DSJE.BADTYPE *InfoType* was unrecognized.
- DSJE.NOTINSTAGE *StageName* was DSJ.ME and the caller is not running within a stage.
- DSJE.BADSTAGE *StageName* does not refer to a known stage in the job.

Remarks

When referring to a controlled job, DSGetStageInfo can be used either before or after a DSRunJob has been issued. Any status returned following a successful call to DSRunJob is guaranteed to relate to that run of the job.

Example

The following command requests the last error message for the loader stage of the job qsales:

```
stage_status = DSGetStageInfo(qsales_handle, "loader",
→ DSJ.STAGELASTERR)
```

DSGetStageLinks

Returns a field mark delimited list containing the names of all of the input/output links of the specified stage.

Syntax

```
Result = DSGetStageLinks(JobName, StageName, Key)
         or
Call DSGetStageLinks(Result, JobName, StageName, Key)
```

JobName is the name of the job in the current project for which information is required. If the *JobName* does not exist in the current project, then the function will return an empty string.

StageName is the name of the stage in the specified job for which information is required. If the *StageName* does not exist in the specified job then the function will return an empty string.

Key depending on the value of Key the returned list will contain all of the stages links (Key=0), only the stage's input links (Key=1) or only the stage's output links (Key=2).

Result returns a field mark delimited list containing the names of the links.

Example

The following returns a list of all the input links on the stage called "join1" in the job "testjob":

```
linkslist = DSGetStageLinks (testjob, join1, 1)
```

DSGetStagesOfType

Returns a field mark delimited list containing the names of all of the stages of the specified type in a named job.

Syntax

```
Result = DSGetStagesOfType (JobName, StageType)
or
Call DSGetStagesOfType (Result, JobName, StageType)
```

JobName is the name of the job in the current project for which information is required. If the JobName does not exist in the current project then the function will return an empty string.

StageType is the name of the stage type, as shown by the repository stage type properties form, such as CTransformerStage or ORAOCI8. If the *StageType* does not exist in the current project or there are no stages of that type in the specified job, then the function will return an empty string.

Result returns a field mark delimited list containing the names of all of the stages of the specified type in a named job.

Example

The following returns a list of all the Aggregator stages in the parallel job "testjob":

```
stagelist = DSGetStagesOfType (testjob, PxAggregator)
```

DSGetStagesTypes

Returns a field mark delimited string of all active and passive stage types that exist within a named job.

Syntax

```
Result = DSGetStageTypes(JobName )
or
Call DSGetStageTypes(Result, JobName )
```

JobName is the name of the job in the current project for which information is required. If *JobName* does not exist in the current project, *Result* will be set to an empty string.

Result is a sorted, field mark delimited string of stage types within *JobName*.

Example

The following returns a list of all the types of stage in the job "testjob":

```
stagetypelist = DSGetStagesOfType (testjob)
```

DSGetVarInfo

Provides a method of obtaining information about variables used in transformer stages.

Syntax

```
Result = DSGetVarInfo (JobHandle, StageName, VarName, InfoType)
```

JobHandle is the handle for the job as derived from DSAttachJob, or it might be DSJ.ME to refer to the current job.

StageName is the name of the stage to be interrogated. It might also be DSJ.ME to refer to the current stage if necessary.

VarName is the name of the variable to be interrogated.

InfoType specifies the information required and can be one of:

DSJ.VARVALUE

DSJ.VARDESCRIPTION

Result depends on the specified *InfoType*, as follows:

- DSJ.VARVALUE String - the value of the specified variable.
- DSJ.VARDESCRIPTION String - description of the specified variable.

Result might also return an error condition as follows:

- DSJE.BADHANDLE *JobHandle* was invalid.
- DSJE.BADTYPE *InfoType* was not recognized.
- DSJE.NOTINSTAGE *StageName* was DSJ.ME and the caller is not running within a stage.
- DSJE.BADVAR *VarName* was not recognized.
- DSJE.BADSTAGE *StageName* does not refer to a known stage in the job.

DSIPCPageProps

Returns the size (in KB) of the Send/Receive buffer of an IPC (or Web Service) stage.

Syntax

```
Result = DSGetIPCStageProps (JobName, StageName)
or
Call DSGetIPCStageProps (Result, JobName, StageName)
```

JobName is the name of the job in the current project for which information is required. If *JobName* does not exist in the current project, *Result* will be set to an empty string.

StageName is the name of an IPC stage in the specified job for which information is required. If *StageName* does not exist, or is not an IPC stage within *JobName*, Result will be set to an empty string.

Result is an array containing the following fields:

- the size (in kilobytes) of the Send/Receive buffer of the IPC (or Web Service) stage StageName within JobName.
- the seconds timeout value of the IPC (or Web Service) stage StageName within JobName.

Example

The following returns the size and timeout of the stage "IPC1" in the job "testjob":

```
buffersize = DSGetIPCStageProps (testjob, IPC1)
```

DSLogEvent

Logs an event message to a job other than the current one. (Use DSLogInfo, DSLogFatal, or DSLogWarn to log an event to the current job.)

Syntax

```
ErrCode = DSLogEvent (JobHandle, EventType, EventMsg)
```

JobHandle is the handle for the job as derived from DSAttachJob.

EventType is the type of event logged and is one of:

- DSJ.LOGINFO Information message
- DSJ.LOGWARNING Warning message

EventMsg is a string containing the event message.

ErrCode is 0 if there is no error. Otherwise it contains one of the following errors:

- DSJE.BADHANDLE Invalid *JobHandle*.
- DSJE.BADTYPE Invalid *EventType* (particularly note that you cannot place a fatal message in another job's log).

Example

The following command, when included in the msales job, adds the message "monthly sales complete" to the log for the qsales job:

```
Logerror = DsLogEvent (qsales_handle, DSJ.LOGINFO,
→ "monthly sales complete")
```

DSLogFatal

Logs a fatal error message in a job's log file and terminates the job.

Syntax

```
Call DSLogFatal (Message, CallingProgName)
```

Message (input) is the warning message you want to log. *Message* is automatically prefixed with the name of the current stage and the calling before/after subroutine.

CallingProgName (input) is the name of the before/after subroutine that calls the DSLogFatal subroutine.

Remarks

DSLogFatal writes the fatal error message to the job log file and aborts the job. DSLogFatal never returns to the calling before/after subroutine, so it should be used with caution. If a job stops with a fatal error, it must be reset by using the Director client before it can be rerun.

In a before/after subroutine, it is better to log a warning message (using DSLogWarn) and exit with a nonzero error code, which allows InfoSphere DataStage to stop the job cleanly.

DSLogFatal should not be used in a transform. Use DSTransformError instead.

Example

```
Call DSLogFatal("Cannot open file", "MyRoutine")
```

DSLogInfo

Logs an information message in a job's log file.

Syntax

```
Call DSLogInfo (Message, CallingProgName)
```

Message (input) is the information message you want to log. *Message* is automatically prefixed with the name of the current stage and the calling program.

CallingProgName (input) is the name of the transform or before/after subroutine that calls the DSLogInfo subroutine.

Remarks

DSLogInfo writes the message text to the job log file as an information message and returns to the calling routine or transform. If DSLogInfo is called during the test phase for a newly created routine in the repository, the two arguments are displayed in the results window.

Unlimited information messages can be written to the job log file. However, if a lot of messages are produced, the job might run slowly and the Director client might take some time to display the job log file.

Example

```
Call DSLogInfo("Transforming: ":Arg1, "MyTransform")
```

DSLogToController

This routine might be used to put an info message in the log file of the job controlling this job, if any. If there isn't one, the call is just ignored.

Syntax

```
Call DSLogToController(MsgString)
```

MsgString is the text to be logged. The log event is of type Information.

Remarks

If the current job is not under control, a silent exit is performed.

Example

```
Call DSLogToController("This is logged to parent")
```

DSLogWarn

Logs a warning message in a job's log file.

Syntax

```
Call DSLogWarn (Message, CallingProgName)
```

Message (input) is the warning message you want to log. *Message* is automatically prefixed with the name of the current stage and the calling before/after subroutine.

CallingProgName (input) is the name of the before/after subroutine that calls the DSLogWarn subroutine.

Remarks

DSLogWarn writes the message to the job log file as a warning and returns to the calling before/after subroutine. If the job has a warning limit defined for it, when the number of warnings reaches that limit, the call does not return and the job is aborted.

DSLogWarn should not be used in a transform. Use DSTransformError instead.

Example

```
If InputArg > 100 Then
   Call DSLogWarn("Input must be =< 100; received
      ":InputArg,"MyRoutine")
End Else
   * Carry on processing unless the job aborts
End
```

DSMakeJobReport

Generates a report describing the complete status of a valid attached job.

Syntax

```
ReportText = DSMakeJobReport(JobHandle, ReportLevel, LineSeparator)
```

JobHandle is the string as returned from DSAttachJob.

ReportLevel specifies the type of report and is one of the following:

- 0 - basic report. Text string containing start/end time, time elapsed and status of job.
- 1 - stage/link detail. As basic report, but also contains information about individual stages and links within the job.
- 2 - text string containing full XML report.

By default the generated XML will not contain a <?xml-stylesheet?> processing instruction. If a stylesheet is required, specify a ReportLevel of 2 and append the name of the required stylesheet URL, that is, 2;styleSheetURL. This inserts a processing instruction into the generated XML of the form:

```
<?xml-stylesheet type=text/xsl" href="styleSheetURL"?>
```

LineSeparator is the string used to separate lines of the report. Special values recognized are:

- "CRLF" => CHAR(13):CHAR(10)

- "LF" => CHAR(10)
- "CR" => CHAR(13)

The default is CRLF if on Windows, else LF.

Remarks

If a bad job handle is given, or any other error is encountered, information is added to the ReportText.

Example

```
h$ = DSAttachJob("MyJob", DSJ.ERRNONE)
rpt$ = DSMakeJobReport(h$,0,"CRLF")
```

DSMakeMsg

Insert arguments into a message template. Optionally, it will look up a template ID in the standard InfoSphere DataStage message file, and use any returned message template instead of that given to the routine.

Syntax

```
FullText = DSMakeMsg(Template, ArgList)
```

FullText is the message with parameters substituted

Template is the message template, in which %1, %2 and so on are to be substituted with values from the equivalent position in *ArgList*. If the template string starts with a number followed by "\", that is assumed to be part of a message id to be looked up in the InfoSphere DataStage message file.

Note: If an argument token is followed by "[E]", the value of that argument is assumed to be a job control error code, and an explanation of it will be inserted in place of "[E]". (See the DSTranslateCode function.)

ArgList is the dynamic array, one field per argument to be substituted.

Remarks

This routine is called from job control code created by the JobSequence Generator. It is basically an interlude to call DSRMessage which hides any runtime includes.

It will also perform local job parameter substitution in the message text. That is, if called from within a job, it looks for substrings such as "#xyz#" and replaces them with the value of the job parameter named "xyz".

Example

```
t$ = DSMakeMsg("Error calling DSAttachJob(%1)<L>%2",
→jb$:@FM:DSGetLastErrorMsg())
```

DSPrepareJob

Used to ensure that a compiled job is in the correct state to be run or validated.

Syntax

```
JobHandle = DSPrepareJob(JobHandle)
```

JobHandle is the handle, as returned from DSAttachJob(), of the job to be prepared.

JobHandle is either the original handle or a new one. If returned as 0, an error occurred and a message is logged.

Example

```
h$ = DSPrepareJob(h$)
```

DSRunJob

Starts a job running. Note that this call is asynchronous; the request is passed to the runtime engine, but you are not informed of its progress.

Syntax

```
ErrCode = DSRunJob (JobHandle, RunMode)
```

JobHandle is the handle for the job as derived from DSAttachJob.

RunMode is the name of the mode that the job is to be run in and is one of:

- DSJ.RUNNORMAL (Default) Standard job run.
- DSJ.RUNRESET Job is to be reset.
- DSJ.RUNVALIDATE Job is to be validated only.
- DSJ.RUNRESTART Restartable job sequence is to be restarted with the original job parameter values.

ErrCode is 0 if DSRunJob is successful, otherwise it is one of the following negative integers:

- DSJE.BADHANDLE Invalid *JobHandle*.
- DSJE.BADSTATE Job is not in the right state (compiled, not running).
- DSJE.BADTYPE *RunMode* is not a known mode.

Remarks

If the controlling job is running in validate mode, then any calls of DSRunJob will act as if *RunMode* was DSJ.RUNVALIDATE, regardless of the actual setting.

A job in validate mode will run its JobControl routine (if any) rather than just check for its existence, as is the case for before/after routines. This allows you to examine the log of what jobs it started up in validate mode.

After a call of DSRunJob, the controlled job's handle is unloaded. If you require to run the same job again, you must use DSDetachJob and DSAttachJob to set a new handle. Note that you will also need to use DSWaitForJob, as you cannot attach to a job while it is running.

Example

The following command starts the job qsales in standard mode:

```
RunErr = DSRunJob(qsales_handle, DSJ.RUNNORMAL)
```

DSSendMail

This routine is an interface to a sendmail program that is assumed to exist somewhere in the search path of the current user (on the engine tier host). It hides the different call interfaces to various sendmail programs, and provides a simple interface for sending text. For example:

Syntax

```
Reply = DSSendMail(Parameters)
```

Parameters is a set of name:value parameters, separated by either a mark character or "\n".

Currently recognized names (case-insensitive) are:

- "From" Mail address of sender, for example, Me@SomeWhere.com
  Can only be left blank if the local template file does not contain a "%from%" token.
- "To" Mail address of recipient, for example, You@ElseWhere.com
  Can only be left blank if the local template file does not contain a "%to%" token.
- "Subject" Something to put in the subject line of the message.
  Refers to the "%subject%" token. If left as "", a standard subject line will be created, along the lines of "From InfoSphere DataStage job: jobname"
- "Server" Name of host through which the mail should be sent.
  might be omitted on systems (such as Unix) where the SMTP host name can be and is set up externally, in which case the local template file presumably will not contain a "%server%" token.
- "Body" Message body.
  Can be omitted. An empty message will be sent. If used, it must be the last parameter, to allow for getting multiple lines into the message, using "\n" for line breaks. Refers to the "%body%" token.

  Note: The text of the body might contain the tokens "%report% or %fullreport% anywhere within it, which will cause a report on the current job status to be inserted at that point. A full report contains stage and link information as well as job status.

Reply. Possible replies are:

- DSJE.NOERROR (0) OK
- DSJE.NOPARAM Parameter name missing - field does not look like 'name:value'
- DSJE.NOTEMPLATE Cannot find template file
- DSJE.BADTEMPLATE Error in template file

Remarks

The routine looks for a local file, in the current project directory, with a well-known name. That is, a template to describe exactly how to run the local sendmail command.

Example

```
code = DSSendMail("From:me@here\nTo:You@there\nSubject:Hi ya\nBody:Line1\nLine2")
```

DSSetDisableJobHandler

Enable or disable job-level message handling.

Syntax

```
ErrCode = DSSetDisableJobHandler (JobHandle, value)
```

JobHandle is the handle for the job as derived from DSAttachJob.

value is TRUE to disable job-level message handling, or FALSE to enable job-level message handling.

ErrCode is 0 if DSSetDisableJobHandler is successful, otherwise it is one of the following negative integers:

- DSJE.BADHANDLE Invalid *JobHandle.*
- DSJE.BADVALUE *value* is not appropriate for that parameter type.

Example

The following command disables job-level message handling for the qsales job:

```
GenErr = DSSetDisableJobHandler (qsales_handle, TRUE)
```

DSSetDisableProjectHandler

Enable or disable project-level message handling.

Syntax

```
ErrCode = DSSetDisableProjectHandler (ProjectHandle, value)
```

ProjectHandle is the value returned from DSOpenProject.

value is TRUE to disable project-level message handling, or FALSE to enable project-level message handling.

ErrCode is 0 if DSSetDisableProjectHandler is successful, otherwise it is one of the following negative integers:

- DSJE.BADHANDLE Invalid *ProjectHandle.*
- DSJE.BADVALUE *value* is not appropriate for that parameter type.

Example

The following command disables project-level message handling for the qsales project:

```
GenErr = DSSetDisableProjectHandler (qsales_handle, TRUE)
```

DSSetGenerateOpMetaData

Use this to specify whether the job generates operational metadata or not. This overrides the default setting for the project.

Syntax

```
ErrCode = DSSetGenerateOpMetaData (JobHandle, value)
```

JobHandle is the handle for the job as derived from DSAttachJob.

value is TRUE to generate operational metadata, FALSE to not generate operational metadata.

ErrCode is 0 if DSRunJob is successful, otherwise it is one of the following negative integers:

- DSJE.BADHANDLE Invalid *JobHandle*.
- DSJE.BADTYPE *value* is wrong.

Example

The following command causes the job qsales to generate operational metadata whatever the project default specifies:

GenErr = DSSetGenerateOpMetaData(qsales_handle, TRUE)

DSSetJobLimit

By default a controlled job inherits any row or warning limits from the controlling job. These can, however, be overridden using the DSSetJobLimit function.

Syntax

```
ErrCode = DSSetJobLimit (JobHandle, LimitType, LimitValue)
```

JobHandle is the handle for the job as derived from DSAttachJob.

LimitType is the name of the limit to be applied to the running job and is one of:

- DSJ.LIMITWARN Job to be stopped after *LimitValue* warning events.
- DSJ.LIMITROWS Stages to be limited to *LimitValue* rows.

LimitValue is an integer specifying the value to set the limit to. Set this to 0 to specify unlimited warnings.

ErrCode is 0 if DSSetJobLimit is successful, otherwise it is one of the following negative integers:

- DSJE.BADHANDLE Invalid *JobHandle.*
- DSJE.BADSTATE Job is not in the right state (compiled, not running).
- DSJE.BADTYPE *LimitType* is not a known limiting condition.
- DSJE.BADVALUE *LimitValue* is not appropriate for the limiting condition type.

Example

The following command sets a limit of 10 warnings on the qsales job before it is stopped:

```
LimitErr = DSSetJobLimit(qsales_handle,
→ DSJ.LIMITWARN, 10)
```

DSSetParam

Specifies job parameter values before running a job. Any parameter not set will be defaulted.

Syntax

```
ErrCode = DSSetParam (JobHandle, ParamName, ParamValue)
```

JobHandle is the handle for the job as derived from DSAttachJob.

ParamName is a string giving the name of the parameter.

ParamValue is a string giving the value for the parameter.

ErrCode is 0 if DSSetParam is successful, otherwise it is one of the following negative integers:

- DSJE.BADHANDLE Invalid *JobHandle.*
- DSJE.BADSTATE Job is not in the right state (compiled, not running).
- DSJE.BADPARAM *ParamName* is not a known parameter of the job.
- DSJE.BADVALUE *ParamValue* is not appropriate for that parameter type.

Example

The following commands set the quarter parameter to 1 and the startdate parameter to 1/1/97 for the qsales job:

```
paramerr = DSSetParam (qsales_handle, "quarter", "1")
paramerr = DSSetParam (qsales_handle, "startdate",
→ "1997-01-01")
```

DSSetUserStatus

Applies only to the current job, and does not take a *JobHandle* parameter. It can be used by any job in either a JobControl or After routine to set a termination code for interrogation by another job. In fact, the code might be set at any point in the job, and the last setting is the one that will be picked up at any time. So to be certain of getting the actual termination code for a job the caller should use DSWaitForJob and DSGetJobInfo first, checking for a successful finishing status.

This routine is defined as a subroutine not a function because there are no possible errors.

Syntax

```
Call DSSetUserStatus (UserStatus)
```

UserStatus String is any user-defined termination message. The string will be logged as part of a suitable "Control" event in the calling job's log, and stored for retrieval by DSGetJobInfo, overwriting any previous stored string.

This string should not be a negative integer, otherwise it might be indistinguishable from an internal error in DSGetJobInfo calls.

Example

The following command sets a termination code of "sales job done":

```
Call DSSetUserStatus("sales job done")
```

DSStopJob

This routine should only be used after a DSRunJob has been issued. It immediately sends a stop request to the runtime engine. The call is asynchronous. If you need to know that the job has actually stopped, you must call DSWaitForJob or use the Sleep statement and poll for DSGetJobStatus. Note that the stop request gets sent regardless of the job's current status.

Syntax

```
ErrCode = DSStopJob (JobHandle)
```

JobHandle is the handle for the job as derived from DSAttachJob.

ErrCode is 0 if DSStopJob is successful, otherwise it might be the following:

- DSJE.BADHANDLE Invalid *JobHandle*.

Example

The following command requests that the qsales job is stopped:

```
stoperr = DSStopJob(qsales_handle)
```

DSTransformError

Logs a warning message to a job log file. This function is called from transforms only.

Syntax

```
Call DSTransformError (Message, TransformName)
```

Message (input) is the warning message you want to log. *Message* is automatically prefixed with the name of the current stage and the calling transform.

TransformName (input) is the name of the transform that calls the DSTransformError subroutine.

Remarks

DSTransformError writes the message (and other information) to the job log file as a warning and returns to the transform. If the job has a warning limit defined for it, when the number of warnings reaches that limit, the call does not return and the job is aborted.

In addition to the warning message, DSTransformError logs the values of all columns in the current rows for all input and output links connected to the current stage.

Example

```
Function MySqrt(Arg1)
If Arg1 < 0 Then
   Call DSTransformError("Negative value:"Arg1, "MySqrt")
   Return("0")    ;*transform produces 0 in this case
End
   Result = Sqrt(Arg1) ;* else return the square root
Return(Result)
```

DSTranslateCode

Converts a job control status or error code into an explanatory text message.

Syntax

```
Ans = DSTranslateCode(Code)
```

Code is:

- If Code > 0, it's assumed to be a job status.
- If Code < 0, it's assumed to be an error code.
- (0 should never be passed in, and will return "no error")

Ans is the message associated with the code.

Remarks

If *Code* is not recognized, then *Ans* will report it.

Example

```
code$ = DSGetLastErrorMsg()
ans$ = DSTranslateCode(code$)
```

DSWaitForFile

Suspend a job until a named file either exists or does not exist.

Syntax

```
Reply = DSWaitForFile(Parameters)
```

Parameters is the full path of file to wait on. No check is made as to whether this is a reasonable path (for example, whether all directories in the path exist). A path name starting with "-", indicates a flag to check the nonexistence of the path. It is not part of the path name.

Parameters might also end in the form " timeout:NNNN" (or "timeout=NNNN") This indicates a non-default time to wait before giving up. There are several possible formats, case-insensitive:

- nnn number of seconds to wait (from now)
- nnnS ditto
- nnnM number of minutes to wait (from now)
- nnnH number of hours to wait (from now)
- nn:nn:nn wait until this time in 24HH:NN:SS. If this or nn:nn time has passed, will wait till next day.

The default timeout is the same as "12H".

The format might optionally terminate "/nn", indicating a poll delay time in seconds. If omitted, a default poll time is used.

Reply might be:

- DSJE.NOERROR (0) OK - file now exists or does not exist, depending on flag.
- DSJE.BADTIME Unrecognized Timeout format
- DSJE.NOFILEPATH File path missing
- DSJE.TIMEOUT Waited too long

Examples

```
Reply = DSWaitForFile("C:\ftp\incoming.txt timeout:2H")
```

(wait 7200 seconds for file on C: to exist before it gives up.)

```
Reply = DSWaitForFile("-incoming.txt timeout=15:00")
```

(wait until 3 p.m. for file in local directory to NOT exist.)

```
Reply = DSWaitForFile("incoming.txt timeout:3600/60")
```

(wait 1 hour for a local file to exist, looking once a minute.)

DSWaitForJob

This function is only valid if the current job has issued a DSRunJob on the given *JobHandle*(s). It returns if the/a job has started since the last DSRunJob has since finished.

Syntax

```
ErrCode = DSWaitForJob (JobHandle)
```

JobHandle is the string returned from DSAttachJob. If commas are contained, it's a comma-delimited set of job handles, representing a list of jobs that are all to be waited for.

ErrCode is 0 if no error, else possible error values (<0) are:

- DSJE.BADHANDLE Invalid JobHandle.
- DSJE.WRONGJOB Job for this JobHandle was not run from within this job.

ErrCode is >0 => handle of the job that finished from a multi-job wait.

Remarks

DSWaitForJob will wait for either a single job or multiple jobs.

Example

To wait for the return of the qsales job:

```
WaitErr = DSWaitForJob(qsales_handle)
```

Dtx Function

Converts a decimal integer to hexadecimal.

Syntax

Dtx (*number* [,*size*])

number is the decimal number to be converted. If *number* is a null value, null is returned.

size is the minimum number of characters in the hexadecimal value. The returned value is padded with leading zeros as required. If *size* is a null value, a runtime error occurs.

Example

This is an example of the **Dtx** function used to convert a decimal number to a hexadecimal string representation:

```
MyNumber = 47
MyHex = Dtx(MyNumber)               ;* returns "2F"
MyHex = Dtx(MyNumber, 4)               ;* returns "002F"
```

Ebcdic Function

Converts the values of characters in a string from ASCII to EBCDIC format.

Syntax

Ebcdic (*string*)

string is the string or expression that you want to convert. If *string* is a null value, a runtime error occurs.

Remarks

The **Ebcdic** and **Ascii** functions perform complementary operations.

Note: If NLS is enabled, this function might return data that is not recognized by the current character set map.

Example

This example shows the **Ebcdic** function being used to convert a string of ASCII bytes:

```
AsciiStr = "A1"
EbcdicStr = Ebcdic(AsciiStr)               ;* convert all bytes to EBCDIC
* (Letter A is decimal 193, digit 1 is decimal 241 in EBCDIC)
If EbcdicStr = Char(193):Char(241) Then
...               ;* ... so this branch is taken
EndIf
```

End Statement

Indicates the end of a program, a subroutine, or a block of statements.

Syntax

```
End
End Case
```

Remarks

Use an **End** statement in the middle of a program to end a section of an **If** statement or other conditional statements.

Use **End Case** to end a set of **Case** statements.

Examples

This example illustrates the use of an **End** statement with various forms of **If...Then** construction in a routine:

```
Function MyTransform(Arg1, Arg2, Arg3)
* Then and Else clauses occupying a single line each:
   If Arg1 Matches "A..."
      Then Reply = 1
      Else Reply = 2
* Multi-line clauses:
   If Len(arg1) > 10 Then
   Reply += 1
   Reply = Arg2 * Reply
   End Else
      Reply += 2
      Reply = (Arg2 - 1) * Reply
   End
* Another style of multiline clauses:
   If Len(Arg1) > 20
   Then
      Reply += 2
      Reply = Arg3 * Reply
   End
   Else
      Reply += 4
      Reply = (Arg3 - 1) * Reply
   End
Return(Reply)
```

This example uses an **End Case** statement with a **Case** statement:

```
Function MyTransform(Arg1)
   Begin Case
   Case Arg1 = 1
      Reply = "A"
   Case Arg1 = 2
      Reply = "B"
   Case Arg1 > 2 And Arg1 < 11
      Reply = "C"
   Case @True      ;* all other values
      Call DSTransformError("Bad arg":Arg1, "MyTransform"
      Reply = ""
   End Case
Return(Reply)
```

Equate Statement

Equates a value to a symbol or a literal string during compilation. Not available in expressions.

Syntax

```
Equate symbol To value [ ,symbol To value] ...
Equate symbol Literally value [ ,symbol Literally value] ...
```

symbol is the equate name you want to give to a value in your program. *symbol* must not be a number.

value is the value you want to identify by *symbol*. *value* must be quoted.

To specifies that value is any type of expression.

Literally (or **Lit**) specifies that value is a literal string.

Remarks

You can equate *symbol* only once, otherwise you get a compiler error.

Example

The following example illustrates the use of both **Equate...To** and **Equate...Literally** to set symbols in code:

```
Function MyFunction(Arg1, Arg2)
Equate Option1 To "O1"
Equate Option2 To "O2"
Equate TestOption Literally "If Arg1 = "
TestOption Option1 Then  ;* code becomes: If Arg1 = "1 Then
   Ans = ...
End
TestOption Option2 Then  ;* code becomes: If Arg1 = "O2" Then
   Ans = ...
End
Return(Ans)
```

Ereplace Function

Replaces one or more instances of a substring.

Syntax

```
Ereplace (string, substring, replacement [ ,number [ ,start] ] )
```

string is the string or expression.

substring is the substring you want to replace. If *substring* is an empty string, the value of *string* is returned.

replacement is the replacement substring. If *replacement* is an empty string, all occurrences of *substring* are removed.

number specifies the number of instances of *substring* to replace. To change all instances, use a value less than 1.

start specifies the first instance to replace. A value less than 1 defaults to 1.

Remarks

A null value for *string* returns a null value. If you use a null value for any other variable, a runtime error occurs.

Examples

The following example replaces all occurrences of one substring with another:

```
MyString = "AABBCCBBDDBB"
NewString = Ereplace(MyString, "BB", "xxx")
* The result is "AAxxxCCxxxDDxxx"
```

The following example replaces only the first two occurrences:

```
MyString = "AABBCCBBDDBB"
NewString = Ereplace(MyString, "BB", "xxx", 2, 1)
* The result is "AAxxxCCxxxDDBB"
```

The following example removes all occurrences of the substring:

```
MyString = "AABBCCBBDDBB"
NewString = Ereplace(MyString, "BB", "")
* The result is "AACCDD"
```

Exchange Function

Replaces a character in a string.

Syntax

Exchange (*string*, *find.character*, *replace.character*)

string is the string or expression containing the character to replace. A null string returns a null.

find.character is the hexadecimal value of the character to find. If *find.character* is a null value, **Exchange** fails and generates a runtime error.

replace.character is the hexadecimal value of the replacement character. If the value of *replacement.character* is FF, *find.character* is deleted from the string. If *replace.character* is a null value, **Exchange** fails and generates a runtime error.

Remarks

Exchange replaces all occurrences of the specified character.

If NLS is enabled, **Exchange** uses the first two bytes of *find.character* and *replace.character*. Characters are evaluated as follows:

Bytes	Evaluated as...
00 through FF	00 through FF
00 through FA	Unicode characters 0000 through FA
FB through FE	System delimiters

Example

In the following example, 41 is the hexadecimal value for the character "A" and 2E is the hexadecimal value for the period (.) character:

```
MyString = Exchange("ABABC", "41", "2E")
* result is ".B.BC"
* The above line is functionally equivalent to:
* MyString = Convert("A", ".", "ABABC")
```

Exp Function

Returns the value of "e" raised to the specified power.

Syntax

Exp (*power*)

power is a number or numeric expression specifying the power. A null value returns a null value. If *power* is too large or too small, a warning message is generated and 0 is returned.

Remarks

The value of "e" is approximately 2.71828. The formula used to perform the calculation is:

Exp *function value* = 2.71828**(*power*)

Example

This example uses the **Exp** function to return "e" raised to a power:

```
* Define angle in radians.
MyAngle = 1.3
* Calculate hyperbolic secant.
MyHSec = 2 / (Exp(MyAngle) + Exp(-MyAngle))
```

Field Function

Returns delimited substrings in a string.

Syntax

Field (*string***,** *delimiter***,** *instance* [**,***number*] **)**

string is the string containing the substring. If *string* is a null value, null is returned.

delimiter is the character that delimits the substring. If *delimiter* is an empty string, *string* is returned. If *string* does not contain *delimiter*, an empty string is returned unless *instance* is 1, in which case *string* is returned. If *delimiter* is a null value, a runtime error occurs. If more than one substring is returned, delimiters are returned with the substrings.

instance specifies which instance of *delimiter* terminates the substring. If *instance* is less than 1, 1 is assumed. If *string* does not contain *instance*, an empty string is returned. If *instance* is a null value, a runtime error occurs.

number specifies the number of delimited substrings to return. If *number* is an empty string or less than 1, 1 is assumed. If *number* is a null value, a runtime error occurs.

Examples

In the following example the variable MyString is set to the data between the third and fourth occurrences of the delimiter "#":

```
MyString = Field("###DHHH#KK","#", 4)  ;* returns "DHHH"
```

In the following example SubString is set to "" since the delimiter "/" does not appear in the string:

```
MyString = "London+0171+NW2+AZ"
SubString = Field(Mystring, "/", 1)    ;* returns ""
```

In the following example SubString is set to "0171+NW2" since two fields were requested using the delimiter "+" (the second and third fields):

```
MyString = "London+0171+NW2+AZ"
SubString = Field(Mystring, "+", 2, 2)
* returns "0171+NW2"
```

FieldStore Function

Modifies character strings by inserting, deleting, or replacing fields separated by specified delimiters.

Syntax

FieldStore (*string*, *delimiter*, *start*, *number*, *new.fields***)**

string is the string to be modified. If *string* is a null value, null is returned.

delimiter delimits the fields and can be any single ASCII character. If *delimiter* is null, there is a runtime error.

start is the number of the field to start the modification.

- If *start* is greater than the number of fields in *string*, the string is padded with empty fields before processing begins.
- If *start* is null, there is a runtime error.

number is the number of fields of *new.fields* to insert in *string*.

- If *number* is positive, *number* fields in *string* are replaced with the first *number* fields of *new.field*s.
- If *number* is negative, *number* fields in *string* are replaced with all the fields in *new.fields*.
- If *number* is 0, all the fields in *new.fields* are inserted in *string* before the field specified by *start*.
- If *number* is null, there is a runtime error.

new.fields is a string or expression containing the new fields to use. If *new.fields* is null, there is a runtime error.

Example

The following examples show several different ways of replacing substrings within a string:

```
MyString = "1#2#3#4#5"
String = Fieldstore(MyString, "#", 2, 2, "A#B")
* Above results in: "1#A#B#4#5"
String2 = Fieldstore(MyString, "#", 2, -2, "A#B")
* Above results in: "1#A#B#4#5"
String3 = Fieldstore(MyString, "#", 2, 0, "A#B")
* Above results in: "1#A#B#2#3#4#5"
String4 = Fieldstore(MyString, "#", 1, 4, "A#B#C#D")
* Above results in: "A#B#C#D#5"
String5 = Fieldstore(MyString, "#", 7, 3, "A#B#C#D")
* Above results in: "1#2#3#4#5##A#B#"
```

FIX Function

Use the FIX function to convert a numeric value to a floating-point number with a specified precision. FIX lets you control the accuracy of computation by eliminating excess or unreliable data from numeric results. For example, a bank application that computes the interest accrual for customer accounts does not need to deal with credits expressed in fractions of cents. An engineering application needs to throw away digits that are beyond the accepted reliability of computations.

Syntax

```
FIX (number [ ,precision [ ,mode ] ] )
```

number is an expression that evaluates to the numeric value to be converted. If *number* evaluates to the null value, null is returned.

precision is an expression that evaluates to the number of digits of precision in the floating-point number. The default precision is 4.

mode is a flag that specifies how excess digits are handled. If *mode* is either 0 or not specified, excess digits are rounded off. If *mode* is anything other than 0, excess digits are truncated.

Examples

The following example calculates a value to the default precision of 4:

```
REAL.VALUE = 37.73629273
PRINT FIX (REAL.VALUE)
```

This is the program output:

```
37.7363
```

The next example calculates the same value to two digits of precision. The first result is rounded off, the second is truncated:

```
PRINT FIX (REAL.VALUE, 2)
PRINT FIX (REAL.VALUE, 2, 1)
```

This is the program output:

```
37.74
37.73
```

Fmt Function

Formats data for output.

Syntax

```
Fmt (string, format)
```

string is the string to be formatted. If *string* is a null value, null is returned.

format is an expression that defines how the string is to be formatted. If *format* is null, the **Fmt** function fails. For detailed syntax, see Format Expression.

Remarks

The format expression provides a pattern for formatting the string. You can specify:

- The length of the output field

- A fill character to pad the field
- Whether the field is right-justified or left-justified
- A numerical, monetary, or date format
- A mask to act as a template for the field

Format Expression

Defines how the string is to be formatted.

Syntax

[*length*] [*fill*] *justification* [*edit*] [*mask*]

Output Length

You specify the number of character positions in the output field using the length parameter. You must specify *length* unless you specify *mask*. (You can specify *length* and *mask*.)

- If *string* is smaller than *length*, it is padded with fill characters.
- If *string* is larger than *length*, the string is divided into fields by a text mark, CHAR(251), inserted every *length* characters. Each field is padded with fill characters to *length*.

Fill Character

You specify the *fill* parameter to define the fill character used to pad the output field to the size specified by *length*. The default fill character is a space. If you want to use a numeric character or the letters L, R, T, or Q as a fill character, you must enclose it in single quotation marks.

Justification

You specify the justification of the output using the justification parameter, which must be one of the following codes:

L Left justify and break at end of field.

R Right justify and break at end of field.

T Left justify and break on space (suitable for text fields).

U Left justify and break on field length.

Monetary and Numeric Formatting

The *edit* parameter lets you specify codes that format a string as numeric or monetary output:

Code	Description
n[*m*]	*n* is a number, 0 through 9, that specifies the number of decimal places to display. If you specify 0 for *n*, the value is rounded to the nearest integer. The output is padded with zeros or rounded to the *n*th decimal place, if required. *m* specifies how to descale the value: • A value of 0 descales the value by the current precision. • A value of 1 through 9 descales the value by *m* minus the current precision. If you do not specify *m*, the default value is 0. The default precision is 4.

$ Prefixes a dollar sign to numeric output.

F Prefixes a franc sign to numeric output.

, Inserts a comma to separate thousands.

Z Suppresses leading zeros. It returns an empty string if the value is 0.

E Surrounds negative numbers with angle brackets.

C Appends cr to negative numbers.

D Appends db to positive numbers.

B Appends db to negative numbers.

M Appends a minus sign to negative numbers.

N Suppresses a minus sign on negative numbers.

T Truncates a number rather than rounding it.

Y If NLS is enabled, prefixes the yen/yuan character to the value.

The **E**, **M**, **C**, **D** and **N** options define numeric representations for monetary use, using prefixes or suffixes. If NLS is enabled, these options override the numeric and monetary conventions set for the current locale.

Masked Output

You can specify a format template for the output using the *mask* parameter. For example, a format pattern of 10L##-##-#### formats the string 31121999 to 31-12-1999. A mask can include any characters, but these characters have special meaning:

#n Specifies that the data is displayed in a field of *n* fill characters. If the format expression does not specify a fill character, a space is used.

%*n* Specifies that the data is displayed in a field of *n* zeros.

**n* Specifies that the data is displayed in a field of *n* asterisks.

Any other characters followed by *n* inserts those characters in the output *n* times.

If you want to use numbers or special characters as literals, you must escape the character with a backslash (\).

mask can be enclosed in parentheses for clarity, in which case you must also parenthesize the whole mask. For example:

((###) ###-####)

The **Status** function returns the result of edit as follows:

0 The edit code is successful.

1 The string or expression is invalid.

2 The edit code is invalid.

Formatting Exponential Numbers

These codes are available for formatting exponential expressions:

Q or QR
Right justify an exponential expression and break on field length.

QL Left justify an exponential expression and break on field length.

*n***E***m* Used with **Q**, **QR**, or **QL** justification, *n* is the number of fractional digits, and *m* specifies the exponent. Each can be a number from 0 through 9.

n.*m* Used with **Q**, **QR**, or **QL** justification, *n* is the number of digits preceding the decimal point, and *m* is the number of fractional digits. Each can be a number from 0 through 9.

Z When used with the **Q** format, only the trailing fractional zeros are suppressed, and a 0 exponent is suppressed.

Examples

The following examples show the effect of various **Fmt** codes. In each case the result is shown as a string so that all significant spaces are visible.

Format Expression
Formatted Value

```
X = Fmt("1234567", "14R2")
       X = "1234567.00"

X = Fmt("1234567", "14R2$,"
       X = " $1,234,567.00"

X = Fmt("12345", "14*R2$,"
       X = "****$12,345.00"

X = Fmt("1234567", "14L2"
       X = "1234567.00"

X = Fmt("0012345", "14R")
       X = "0012345"

X = Fmt("0012345", "14RZ")
       X = "12345"

X = Fmt("00000", "14RZ")
       X = " "

X = Fmt("12345", "14'0'R")
       X = "00000000012345"

X = Fmt("ONE TWO THREE", "10T")
       X = "ONE TWO ":T:"THREE"

X = Fmt("ONE TWO THREE", "10R")
       X = "ONE TWO TH":T:"REE "

X = Fmt("AUSTRALIANS", "5T")
       X = "AUSTR":T:"ALIAN":T:"S "

X = Fmt("89", "R#####")
       X = " 89"

X = Fmt("6179328323", "L###-#######")
       X = "617-9328323"

X = Fmt("123456789", "L#3-#3-#3")
       X = "123-456-789"

X = Fmt("123456789", "R#5")
       X = "56789"

X = Fmt("67890", "R#10")
       X = " 67890"
```

```
X = Fmt("123456789", "L#5")
       X = "12345"
X = Fmt("12345", "L#10")
       X = "12345 "
X = Fmt("123456", "R##-##-##")
       X = "12-34-56"
X = Fmt("555666898", "20*R2$,")
       X = "*****$555,666,898.00"
X = Fmt("DAVID", "10.L")
       X = "DAVID....."
X = Fmt("24500", "10R2$Z")
       X = " $24500.00"
X = Fmt("0.12345678E1", "9*Q")
       X = "*1.2346E0"
X = Fmt("233779", "R")
       X = "233779"
X = Fmt("233779", "R0")
       X = "233779"
X = Fmt("233779", "R00")
       X = "2337790000"
X = Fmt("233779", "R2")
       X = "233779.00"
X = Fmt("233779", "R20")
       X = "2337790000.00"
X = Fmt("233779", "R24")
       X = "233779.00"
X = Fmt("2337.79", "R")
       X = "2337.79"
X = Fmt("2337.79", "R0"
       X = "2338"
X = Fmt("2337.79", "R00")
       X = "23377900"
X = Fmt("2337.79", "R2")
       X = "2337.79"
X = Fmt("2337.79", "R20")
       X = "23377900.00"
X = Fmt("2337.79", "R24")
       X = "2337.79"
X = Fmt("2337.79", "R26")
       X = "23.38"
```

FmtDP Function

In NLS mode, formats data in display positions rather than by character length.

Syntax

FmtDP (*string*, *format* **[,** *mapname***])**

string is the string to be formatted. If *string* is a null value, null is returned. Any unmappable characters in the string are assumed to have a display length of 1.

format is an expression that defines how the string is to be formatted. If *format* is null, **FmtDP** fails. For detailed syntax, see Format Expression.

mapname is the name of a character set map to use for the formatting. If *mapname* is not specified, the current default for the project or job is used.

Remarks

FmtDP is suitable for use with multibyte character sets. If NLS is not enabled, the **FmtDP** function works like an equivalent **Fmt** function.

Fold Function

Folds strings to create substrings.

Syntax

Fold (*string*, *length*)

string is the string to be folded.

length is the length of the substrings in characters.

Remarks

Use the **Fold** function to divide a string into a number of substrings separated by field marks.

string is separated into substrings of length less than or equal to *length*. *string* is separated on blanks, if possible, otherwise it is separated into substrings of the specified length.

If *string* evaluates to the null value, null is returned. If *length* is less than 1, an empty string is returned. If *length* is the null value, **Fold** fails and the program terminates with a runtime error message.

Example

```
A=Fold("This is a folded string", 5)
```

Sets A to:

```
ThisFis a FfoldeFdFstrinFg
```

Where F is the field mark.

FoldDP Function

In NLS mode, folds strings to create substrings using character display positions.

Syntax

FoldDP (*string*, *length* **[,** *mapname* **])**

string is the string to be folded.

length is the length of the substrings in display positions.

mapname is the name of a character set map to use for the formatting. If *mapname* is not specified, the current default for the project or job is used.

Remarks

The **FoldDP** function is suitable for use with multibyte character sets. If NLS is not enabled, the **FoldDP** function works like an equivalent **Fold** function.

For...Next Statements

Create a **For...Next** program loop. Not available in expressions.

Syntax

```
For variable = start To end [Step increment]
   [loop.statements]
   [Continue|Exit]
[ {While | Until} condition]
   [loop.statements]
   [Continue]
Next [variable]
```

For *variable* identifies the start of the loop.

start **To** *end* specifies the start and end value of the counter that defines how many times the program is to loop.

Step *increment* specifies the amount the counter is increased when a **Next** statement is reached.

loop.statements are the statements that are executed in the loop.

Continue starts the next iteration of the loop from a point within the loop.

Exit exits the loop from a point within the loop.

While...Continue is an inner loop. If *condition* evaluates to true, the inner loop continues to execute. When *condition* evaluates to false, the inner loop ends. Program execution continues with the statement following the **Next** statement. If *condition* evaluates to a null value, the condition is false.

Until...Continue is an inner loop. If *condition* evaluates to false, the inner loop continues to execute. When *condition* evaluates to true, the loop ends and program execution continues with the statement following the **Next** statement.

condition defines the condition for executing a **While** or **Until** loop. *condition* can be any statement that takes a **Then...Else** clause, but you do not include a **Then...Else** clause. Instead, when the conditional statement would have executed the **Else** clause, *condition* evaluates to false; when the conditional statement would have executed the **Then** clause, *condition* evaluates to true. The **Locked** clause is not supported in this context.

Next *variable* specifies the end of the loop. *variable* is the variable used to define the loop with the **For** statement. Its use is optional, but is recommended to improve the readability of the program, particularly if you use nested loops.

Remarks

You can use multiple **While** and **Until** clauses in a **For...Next** loop. If you nest **For...Next** loops, each loop must have a unique variable name as its counter. If a **Next** statement has no corresponding **For** statement, it generates a compiler error.

Example

This example uses **For...Next** statements to create a string that contains three instances of the numbers 5 through 1, each string separated from the other by a hyphen. The outer loop uses a loop counter variable that is decremented by 1 each time through the loop.

```
String = ""                   ;* starting value must be set up
For Outer = 5 To 1 Step -1    ;* outer 5 repetitions
   For Inner = 1 To 3         ;* inner 5 repetitions
      String = String : Outer
   Next Inner
String = String : "-"      ;* append a hyphen
Next Outer
* String will now look like: 555-444-333-222-111-.
```

Function Statement

Identifies a user-written function and specifies the number and names of the arguments to be passed to it. Not available in expressions.

Syntax

Function [*name*] [*argument1* [**,** *argument2*] **...**]

name is the name of the user-written function and can be any valid variable name.

argument1 and *argument2* are the formal names of arguments to be passed to the function. The formal names reference the actual names of the parameters that are used in the calling program (see the examples). You can specify up to 254 arguments. The calling function in the main program must specify the same number of arguments as the **Function** statement.

Remarks

A user-written function can contain only one **Function** statement, which must be the first noncomment line.

An extra argument is hidden so that the user-written function can use it to return a value. An extra argument is retained by the user-written function so that a value is returned by the **Return** statement. If you use the **Return** statement in a user-written function and you do not specify a value to return, an empty string is returned.

Calling the User-Written Function

The calling program must contain a **Deffun** statement that defines the user-written function before it is called. The user-written function must be cataloged in either a local catalog or the system catalog, or it must be a record in the same object file as the calling program.

If the user-defined function calls itself recursively, you must include a **Deffun** statement preceding the recursive call. For example:

```
Function Cut(expression, character)
Deffun Cut (A1,A2)
If character # '' Then
```

```
...
   Return (Cut (expression, character [2,999999]))
End Else
   Return (expression)
End
End
```

Examples

In this example, a user-defined function called **Short** compares the length of two arguments and returns the shorter:

```
Function Short(A,B)
AL = Len(A)
BL = Len(B)
If AL < BL Then Result = A Else Result = B
Return(Result)
```

In this example, a function called **MyFunc** is defined with the argument names A, B, and C. It is followed by an example of the **DefFun** statement declaring and using the **MyFunc** function. The values held in X, Y, and Z are referenced by the argument names A, B, and C so that the value assigned to T can be calculated.

```
Function MyFunc(A, B, C)
Z = ...
Return (Z)
   ...
   End
DefFun MyFunc(X, Y, Z)
T = MyFunc(X, Y, Z)
End
```

This example shows how to call a transform function named **MyFunctionB** from within another transform function named **MyFunctionA**:

```
Function MyFunctionA(Arg1)
* When referencing a user-written function that is held in the
* DataStage repository, you must declare it as a function with
* the correct number of arguments, and add a "DSU." prefix.
Deffun MyFunctionB(A) Calling "DSU.MyFunctionB"
   Ans = MyFunctionB(Arg1)
* Add own transformation to the value in Ans...
...
```

GetLocale Function

In NLS mode, retrieves the current locale setting for a specified category.

Syntax

```
$Include UNIVERSE.INCLUDE UVNLSLOC.H
name = GetLocale (category)
```

category is one of the following include tokens:

Token Meaning

UVLC$TIME
Time and date

UVLC$NUMERIC
Numeric

UVLC$MONETARY
Currency

UVLC$CTYPE
Character type

UVLC$COLLATE
Sorting sequence

Remarks

GetLocale returns one of the following error tokens if it cannot retrieve the locale setting:

Error Meaning

LCE$NOLOCALES
NLS is not enabled for IBM InfoSphere DataStage.

LCE$BAD.CATEGORY
The specified category is not recognized.

GoSub Statement

Transfers program control to an internal subroutine. Not available in expressions.

Syntax

GoSub *statement.label* [**:**]

statement.label defines where the subroutine starts, and can be any valid label defined in the program.

: identifies the preceding text as a statement label to make the program more readable.

Remarks

You transfer control back to the main program using either a **Return** or **Return To** statement:

- **Return** transfers program control to the statement following the **GoSub** statement.
- **Return To** *label* transfers program control to a location in the program specified by *label*.

A program can call a subroutine any number of times. You can nest subroutines up to 256 deep.

Example

This example uses **GoSub** to call an internal subroutine within an IBM InfoSphere DataStage transform function. The **Return** statement causes execution to resume at the statement immediately following the **GoSub** statement. It is necessary to use **GoTo** as shown to prevent control from accidentally flowing into the subroutine.

```
Function MyTransform(Arg1)
* Only use subroutine if input is a positive number:
   If Arg1 > 0 Then GoSub MyRoutine
   Reply = Arg1
   GoTo ExitFunction  ;* use GoTo to prevent an error
MyRoutine:
   Arg1 = SQRT(Arg1)  ;* take the square root
   Return             ;* return control to statement
ExitFunction:
Return(Reply)
```

GoTo Statement

Transfers program control to the specified statement. Not available in expressions.

Syntax

```
GoTo statement.label [ : ]
```

statement.label specifies the statement to go to.

: identifies the preceding text as a statement label to make the program more readable.

Remarks

If the referenced statement is executable, it is executed and the program continues. If it is not executable, the program goes on to the first executable statement after the referenced one.

Example

This example uses the **GoTo** statement to branch to line labels within a routine. Note that this sort of processing is often clearer using a **Begin Case** construct.

```
Function MyTransform(Arg1)
* Evaluate argument and branch to appropriate label.
   If Arg1 = 1 Then GoTo Label1 Else GoTo Label2
Label1:
   Reply = "A"
   GoTo LastLabel
Label2:
   Reply = "B"
LastLabel:
Return(Reply)
```

Iconv Function

Converts a string to an internal storage format.

Syntax

```
Iconv (string, code [ @VM code ] ... )
```

string evaluates to the string to be converted. If *string* is a null value, null is returned.

code is a conversion code and must be quoted. Multiple conversion codes must be separated by value marks. Multiple codes are applied from left to right. The second code converts the output of the first, and so on. If *code* is a null value, it generates a runtime error.

Remarks

The **Status** function returns the result of the conversion as follows:

0 The conversion was successful.

1 The string was invalid. An empty string was returned, unless *string* was a null value when null was returned.

2 The conversion was invalid.

3 Successful conversion but the input data might be invalid, for example, a nonexistent date, such as 31 September.

Examples

ASCII Conversions

The following examples show the effect of some **MY** (ASCII) conversion codes:

Conversion Expression
Internal Value

```
X = Iconv("ABCD", "MY")
       X = 41424344

X = Iconv("0123", "MY")
       X = 30313233
```

Date Conversions

The following examples show the effect of various **D** (Date) conversion codes:

Conversion Expression
Internal Value

```
X = Iconv("31 DEC 1967", "D")
       X = 0

X = Iconv("27 MAY 97", "D2")
       X = 10740

X = Iconv("05/27/97", "D2/")
       X = 10740

X = Iconv("27/05/1997", "D/E")
       X = 10740

X = Iconv("1997 5 27", "D YMD")
       X = 10740

X = Iconv("27 MAY 97", "D DMY")
       X = 10740

X = Iconv("5/27/97", "D/MDY")
       X = 10740

X = Iconv("27 MAY 1997", "D DMY")
       X = 10740

X = Iconv("97 05 27", "DYMD")
       X = 10740
```

Group Conversions

The following examples show the effect of some **G** (Group) conversion codes:

Conversion Expression
Internal Value

```
X = Iconv("27.05.1997", "G1.2")
       X = "05.1997"

X = Iconv("27.05.1997", "G.2")
       X = "27.05"
```

Length Conversions

The following examples show the effect of some **L** (Length) conversion codes:

Conversion Expression
Internal Value

```
X = Iconv("QWERTYUIOP", "L0")
       X = 10

X = Iconv("QWERTYUIOP", "L7")
       X = ""

X = Iconv("QWERTYU", "L7")
       X = "QWERTYU"

X = Iconv("QWERTYUOP", "L3,5")
       X = ""

X = Iconv("QWER", "L3,5")
       X = "QWER"
```

Masked Character Conversions

The following examples show the effect of some masked character conversion codes (**MCA**, **MC/A**, **MCD**, **MCL**, **MCN**, **MC/N**, **MCP**, **MCT**, **MCU**, and **MCX**):

Conversion Expression
Internal Value

```
X = Iconv("John Smith 1-234", "MCA")
       X = "JohnSmith"

X = Iconv("John Smith 1-234","MC/A")
       X = " 1-234"

X = Iconv("4D2", "MCD")
       X = "1234"

X = Iconv("4D2", "MCDX")
       X = "1234"

X = Iconv("John Smith 1-234", "MCL")
       X = "john smith 1-234"

X = Iconv("John Smith 1-234", "MCN")
       X = "1234"

X = Iconv("John Smith 1-234", "MC/N")
       X = "John Smith -"

X = Iconv("John^CSmith^X1-234", "MCP")
       X = "John.Smith.1-234"

X = Iconv("john SMITH 1-234", "MCT")
       X = "John Smith 1-234"

X = Iconv("john smith 1-234", "MCU")
       X = "JOHN SMITH 1-234"

X = Iconv("1234", "MCX")
       X = "4D2"

X = Iconv("1234", "MCXD")
       X = "4D2"
```

Masked Decimal Conversions

The following examples show the effect of some MD (Masked Decimal) conversion codes:

Conversion Expression
Internal Value

```
X = Iconv("9876.54", "MD2")
       X = 987654

X = Iconv("987654", "MD0")
       X = 987654

X = Iconv("$1,234,567.89", "MD2$,")
       X = 123456789

X = Iconv("123456.789", "MD33")
       X = 123456789

X = Iconv("12345678.9", "MD32")
       X = 1234567890

X = Iconv("F1234567.89", "MD2F")
       X = 123456789

X = Iconv("1234567.89cr", "MD2C")
       X = -123456789

X = Iconv("1234567.89 ", "MD2D")
       X = 123456789

X = Iconv("1,234,567.89 ", "MD2,D")
       X = 123456789

X = Iconv("9876.54", "MD2-Z")
       X = 987654

X = Iconv("$####1234.56", "MD2$12#")
       X = 123456

X = Iconv("$987.654 ", "MD3,$CPZ")
       X = 987654

X = Iconv("####9,876.54", "MD2,ZP12#")
       X = 987654
```

Masked Left and Right Conversions

The following examples show the effect of some **ML** and **MR** (Masked Left and Right) conversion codes:

Conversion Expression
Internal Value

```
X = Iconv("$1,234,567.89", "ML2$,")
       X = 123456789

X = Iconv(".123", "ML3Z")
       X = 123

X = Iconv("123456.789", "ML33")
       X = 123456789

X = Iconv("12345678.9", "ML32")
       X = 1234567890

X = Iconv("1234567.89cr", "ML2C")
       X = -123456789

X = Iconv("1234567.89db", "ML2D")
       X = 123456789
```

```
X = Iconv("1234567.89-", "ML2M")
       X = -123456789

X = Iconv("<1234567.89>", "ML2E")
       X = -123456789

X = Iconv("1234567.89**", "ML2(*12)")
       X = 123456789

X = Iconv("**1234567.89", "MR2(*12)")
       X = 123456789
```

Numeral Conversions

The following examples show the effect of some **NR** (Roman numeral) conversion codes:

```
Conversion Expression
       Internal Value

X = Iconv("mcmxcvii", "NR")
       X = 1997

X = Iconv("MCMXCVmm", "NR")
       X = 1997000
```

Pattern Matching Conversions

The following examples show the effect of some **P** (Pattern matching) conversion codes:

```
Conversion Expression
       Internal Value

X = Iconv("123456789", "P(3N-3A-3X);(9N)")
       X = "123456789"

X = Iconv("123-ABC-A7G", "P(3N-3A-3X);(9N)")
       X = "123-ABC-A7G"

X = Iconv("123-45-6789", "P(3N-2N-4N)")
       X = "123-45-6789"
```

Radix Conversions

The following examples show the effect of some **MX**, **MO**, and **MB** (Radix) conversion codes:

```
Conversion Expression
       Internal Value

X = Iconv("400", "MX")
       X = 1024

X = Iconv("434445", "MX0C")
       X = "CDE"

X = Iconv("2000", "MO")
       X = 1024

X = Iconv("103104105", "MO0C")
       X = "CDE"

X = Iconv("10000000000", "MB")
       X = 1024

X = Iconv("010000110100010001000101", "MB0C")
       X = "CDE"
```

Range Check Conversions

The following example shows the effect of the **R** (Range check) conversion code:

Conversion Expression
Internal Value

```
X = Iconv("123", "R100,200")
       X = 123
```

Soundex Conversions

The following examples show the effect of some **S** (Soundex) conversion codes:

Conversion Expression
Internal Value

```
X = Iconv("GREEN", "S")
       X = "G650"

X = Iconv("greene", "S")
       X = "G650"

X = Iconv("GREENWOOD", "S")
       X = "G653"

X = Iconv("GREENBAUM", "S")
       X = "G651"
```

Time Conversions

The following examples show the effect of some **MT** (Time) conversion codes:

Conversion Expression
Internal Value

```
X = Iconv("02:46", "MT")
       X = 9960

X = Iconv("02:46:40am", "MTHS")
       X = 10000

X = Iconv("02:46am", "MTH")
       X = 9960

X = Iconv("02.46", "MT.")
       X = 9960

X = Iconv("02:46:40", "MTS")
       X = 10000
```

If...Else Statements

Execute one or more statements conditionally. You can use a single-line syntax or multiple lines in a block. Not available in expressions.

Syntax

```
If condition Else statement
If condition  Else   statements  End
```

condition is a numeric value or comparison whose value determines the program flow. If *condition* is false, the statements are executed.

statements are the statements to be executed when *condition* is false.

Remarks

If you want to execute more than one statement when *condition* is false, use the multiline syntax.

Example

```
Function MyTransform(Arg1, Arg2, Arg3)
* Else clause occupying a single line only:
   Reply = 0               ;* default
   If Arg1 Matches "A..."
   Else Reply = 2
* Multi-line Else clause:
   If Len(arg1) > 10 Else
      Reply += 2
      Reply = (Arg2 - 1) * Reply
   End
* Another style of multiline Else clause:
   If Len(Arg1) > 20
   Else
      Reply += 4
      Reply = (Arg3 - 1) * Reply
   End
Return(Reply)
```

If...Then...Else Statements

Define several blocks of statements and the conditions that determine which block is executed. You can use a single-line syntax or multiple lines in a block. Not available in expressions.

Syntax

```
If condition Then statements [Else statements]
If condition
   Then statements  End   [Else statements  End]
```

condition is a numeric value or comparison whose value determines the program flow. If *condition* is true, the **Then** clause is taken. If *condition* is false, the **Else** clause is taken. If *condition* is a null value, it evaluates to false.

statements are the statements to be executed depending on the value of *condition*.

Remarks

You can nest **If...Then...Else** statements. If the **Then** or **Else** statements are written on more than one line, you must use an **End** statement as the last statement.

Example

```
Function MyTransform(Arg1, Arg2, Arg3)
* Then and Else clauses occupying a single line each:
   If Arg1 Matches "A..."
      Then Reply = 1
   Else Reply = 2
* Multi-line clauses:
   If Len(arg1) > 10 Then
      Reply += 1
      Reply = Arg2 * Reply
   End Else
      Reply += 2
      Reply = (Arg2 - 1) * Reply
   End
```

```
* Another style of multiline clauses:
   If Len(Arg1) > 20
   Then
      Reply += 2
      Reply = Arg3 * Reply
   End
   Else
      Reply += 4
      Reply = (Arg3 - 1) * Reply
   End
Return(Reply)
```

If...Then Statements

Execute one or more statements conditionally. You can use a single-line syntax or multiple lines in a block. Not available in expressions.

Syntax

```
If condition Then statement
If conditionThen
   statementsEnd
```

condition is a numeric value or comparison whose value determines the program flow. If *condition* is true, the statements are executed.

statements are the statements to be executed when condition is true.

Remarks

If you want to execute more than one statement when *condition* is true, use the multiline syntax.

Example

This example illustrates various forms of **If...Then** construction that can be used in a routine:

```
Function MyTransform(Arg1, Arg2, Arg3)
* Then clause occupying a single line only:
   Reply = 0               ;* default
   If Arg1 Matches "A..."
      Then Reply = 1
* Multi-line Then clause:
   If Len(arg1) > 10 Then
      Reply += 1
      Reply = Arg2 * Reply
   End

* Another style of multiline Then clause:
   If Len(Arg1) > 20
   Then
      Reply += 2
      Reply = Arg3 * Reply
   End
Return(Reply)
```

If...Then...Else Operator

Assign a value that meets the specified conditions.

If...Then...Else Operator

Syntax

variable = **If** *condition* **Then** *expression* **Else** *expression*

variable is the variable to assign.

If *condition* defines the condition that determines which value to assign.

Then *expression* defines the value to assign if *condition* is true.

Else *expression* defines the value to assign if *condition* is false.

Remarks

The **If** operator is the only form of **If...Then...Else** construction that can be used in an expression.

Example

Note that the **Else** clause is required.

```
* Return A or B depending on value in Column1:
If Column1 > 100 Then "A" Else "B"
* Add 1 or 2 to value in Column2 depending on what's in
* Column3, and return it:
Column2 + (If Column3 Matches "A..." Then 1 Else 2)
```

Index Function

Returns the starting position of a substring.

Syntax

Index (*string***,** *substring***,** *instance***)**

string is the string or expression containing the substring. If *string* is a null value, 0 is returned.

substring is the substring to be found. If *substring* is an empty string, 1 is returned. If *substring* is a null value, 0 is returned.

instance specifies which instance of *substring* is to be located. If *instance* is not found, 0 is returned. If *instance* is a null value, it generates a runtime error.

Examples

The following examples show several ways of finding the position of a substring within a string:

```
MyString = "P1234XXOO1299XX00P1"
Position = Index(MyString, 1, 2)
* The above returns the index of the second "1" character (10).
Position = Index(MyString, "XX", 2)
* The above returns the start index of the second "XX"
* substring (14).
Position = Index(MyString, "xx", 2)
* The above returns 0 since the substring "xx" does not occur.
Position = Index(MyString, "XX", 3)
* The above returns 0 since the third occurrence of
* substring "XX" * cannot be found.
```

InMat Function

Retrieves the dimensions of an array, or determines if a **Dimension** statement failed due to insufficient memory. Not available in expressions.

Syntax

InMat [**(***array***)**]

array is the name of the array whose dimensions you want to retrieve.

Remarks

If you specify *array*, **InMat** returns the dimensions of the array. If you do not specify *array*, **InMat** returns 1 if the preceding **Dimension** statement failed due to lack of available memory.

Example

This example shows how to test whether a **Dimension** statement successfully allocated enough memory:

```
Dim MyArray(2000)
If InMat() = 1 Then
     Call DSLogFatal("Could not allocate array",
→ "MyRoutine")
End
```

Int Function

Returns the integer portion of a numeric expression.

Syntax

Int (*expression***)**

expression is a numeric expression. After evaluation, the fractional portion of the value is truncated and the integer portion is returned. If *expression* is a null value, null is returned.

Example

This example shows the integer portion of an expression being returned by the **Int** function:

```
MyValue = 2.3
IntValue = Int(MyValue)                ;* answer is 2
IntValue = Int(-MyValue)               ;* answer is -2
IntValue = Int(MyValue / 10)               ;* answer is 0
```

IsNull Function

Tests if a variable contains a null value.

Syntax

IsNull (*variable***)**

variable is the variable to test. If *variable* contains a null value, 1 is returned, otherwise 0 is returned.

Remarks

This is the only way to test for a null value because the null value is not equal to any value, including itself.

Example

This example shows how to test for an expression being set to the null value:

```
MyVar = @Null                    ;* sets variable to null value
If IsNull(MyVar * 10) Then
* Will be true since any arithmetic involving a null value
* results in a null value.
End
```

Left Function

Extracts a substring from the start of a string.

Syntax

Left (*string***, *n*)**

string is the string containing the substring. If *string* is a null value, null is returned.

n is the number of characters to extract from the start of the string. If *n* is a null value, it generates a runtime error.

Examples

These examples extract the leftmost three characters of a string:

```
MyString = "ABCDEF"
MySubStr = Left(MyString, 3)               ;* answer is "ABC"
MySubStr = Left("AB", 3)                  ;* answer is "AB"
```

Len Function

Returns the number of characters in a string.

Syntax

Len (*string***)**

string is the string whose characters are counted. All characters are counted, including spaces and trailing blanks. If *string* is a null value, 0 is returned.

Examples

These examples find the length of a string, or a number when expressed as a string:

```
MyStr = "PORTLAND, OREGON"
StrLen = Len(MyStr)               ;* answer is 16
NumLen = Len(12345.67)               ;* answer is 8 (note
                 ;* decimal point)
```

LenDP Function

In NLS mode, returns the length of a string in display positions.

Syntax

LenDP (*string* [**,** *mapname*]**)**

string is the string to be measured. Any unmappable characters in *string* are assumed to have a display length of 1.

mapname is the name of the map that defines the character set used in *string*. If *mapname* is omitted, the default character set map for the project or job is used.

Remarks

If NLS is not enabled, this function works like the **Len** function and returns the number of characters in the string.

Ln Function

Calculates the natural logarithm of the value of an expression, using base "e".

Syntax

Ln (*expression***)**

expression is the numeric expression to evaluate. If *expression* is 0 or negative, 0 is returned and a warning is issued. If *expression* is a null value, null is returned.

Remarks

The value of "e" is approximately 2.71828.

Example

This example shows how to write a transform to convert a number to its base 10 logarithm using the **Ln** function:

```
Function Log10(Arg1)
   If Not(Num(Arg1)) Then
      Call DSTransformError("Non-numeric ":Arg1, "Log10")
      Ans = 0               ;* or some suitable default
   End Else
      Ans = Ln(Arg1) / Ln(10)
   End
Return(Ans)
```

LOCATE Statement

Use a LOCATE statement to search *dynamic.array* for *expression* and to return a value indicating one of the following:

- Where *expression* was found in *dynamic.array*
- Where *expression* should be inserted in *dynamic.array* if it was not found

The search can start anywhere in *dynamic.array*.

Syntax

```
LOCATE expression IN dynamic.array [ < field# [,value#] > ] [ ,start] [BY seq] SETTING variable
{THEN statements [ELSE statements] | ELSE statements}
```

expression evaluates to the string to be searched for in *dynamic.array*. If *expression* or *dynamic.array* evaluate to the null value, *variable* is set to 0 and the ELSE statements are executed. If *expression* and *dynamic.array* both evaluate to empty strings, *variable* is set to 1 and the THEN statements are executed.

field#, *value#*, and *subvalue#* are delimiter expressions, specifying:

- Where the search is to start in *dynamic.array*
- What kind of element is being searched for

start evaluates to a number specifying the field, value, or subvalue from which to start the search.

The delimiter expressions specify the level of the search, and *start* specifies the starting position of the search.

If any delimiter expression or *start* evaluates to the null value, the LOCATE statement fails and the program terminates with a runtime error message.

variable stores the index of *expression*. *variable* returns a field number, value number, or a subvalue number, depending on the delimiter expressions used. *variable* is set to a number representing one of the following:

- The index of the element containing *expression*, if such an element is found
- An index that can be used in an INSERT function to create a new element with the value specified by *expression*

Remarks

During the search, fields are processed as single-valued fields even if they contain value or subvalue marks. Values are processed as single values, even if they contain subvalue marks.

The search stops when one of the following conditions is met:

- A field containing *expression* is found.
- The end of the dynamic array is reached.
- A field that is higher or lower, as specified by *seq*, is found.

If the elements to be searched are sorted in one of the ascending or descending ASCII sequences listed below, you can use the BY *seq* expression to end the search. The search ends at the place where *expression* should be inserted to maintain the ASCII sequence, rather than at the end of the list of specified elements.

Use the following values for *seq* to describe the ASCII sequence being searched:

"AL" or "A"
Ascending, left-justified (standard alphanumeric sort)

"AR" Ascending, right-justified

"DL" or "D"
Descending, left-justified (standard alphanumeric sort)

"DR" Descending, right-justified

seq does not reorder the elements in *dynamic.array*; it specifies the terminating conditions for the search. If a *seq* expression is used and the elements are not in the sequence indicated by *seq*, an element with the value of *expression* might not be found. If *seq* evaluates to the null value, the statement fails and the program terminates.

The ELSE statements are executed if *expression* is not found. The format of the ELSE statement is the same as that used in the IF...THEN statement.

If NLS is enabled, the LOCATE statement with a BY *seq* expression uses the Collate convention as specified *by the current locale*.

Examples

A field mark is shown by **F**, a value mark is shown by **V**, and a subvalue mark is shown by **S**.

```
Q='X':@SM:"$":@SM:'Y':@VM:'Z':@SM:4:@SM:2:@VM:'B':@VM
PRINT "Q= ":Q
LOCATE "$" IN Q <1> SETTING WHERE ELSE PRINT 'ERROR'
PRINT "WHERE= ",WHERE
LOCATE "$" IN Q <1,1> SETTING HERE ELSE PRINT 'ERROR'
PRINT "HERE= ", HERE
NUMBERS=122:@FM:123:@FM:126:@FM:130:@FM
PRINT "BEFORE INSERT, NUMBERS= ",NUMBERS
NUM= 128
LOCATE NUM IN NUMBERS <2> BY "AR" SETTING X ELSE
NUMBERS = INSERT(NUMBERS,X,0,0,NUM)
PRINT "AFTER INSERT, NUMBERS= ",NUMBERS
END
```

This is the program output:

```
Q= XS$SYVZS4S2VBV
ERROR
WHERE= 5
HERE=  2
BEFORE INSERT, NUMBERS= 122F123F126F130FAFTER INSERT, NUMBERS=  122F128F123F126F130F
```

Loop...Repeat Statements

Define a program loop. Not available in expressions.

Syntax

```
Loop   [statements]
   [Continue | Exit]
[While | Until condition Do]
   [statements]
   [Continue | Exit]
Repeat
```

Loop defines the start of the program loop.

statements are the statements that are executed in the loop.

Continue specifies that the current loop breaks and restarts at this point.

Exit specifies that the program quits from the current loop.

While *condition* **Do** specifies that the loop repeats as long as *condition* is true. When *condition* is false, the loop stops and program execution continues with the statement following the **Repeat** statement. If *condition* is a null value, it is considered false.

Until *condition* **Do** specifies that the loop repeats as long as *condition* is false. When *condition* is true, the loop stops and program execution continues with the statement following the **Repeat** statement. If *condition* is a null value, it is considered false.

Repeat defines the end of the loop.

Remarks

You can use multiple **While** and **Until** clauses in a **Loop...Repeat** loop. You can nest **Loop...Repeat** loops. If a **Repeat** statement does not have a corresponding **Loop** statement, it generates a compiler error.

Example

This example shows how **Loop...Repeat** statements can be used. The inner **Loop...Repeat** statement loops 10 times, sets the value of the flag to false, and exits prematurely using the **Exit** statement. The outer loop exits immediately upon checking the value of the flag.

```
Check = @True
Counter = 0      ;* initialize variables
Loop             ;* outer loop
   Loop While Counter < 20   ;* inner loop
      Counter += 1           ;* increment Counter
      If Counter = 10 Then   ;* if condition is True...
         Check = @False      ;* set value of flag to False...
         Exit                ;* and exit from inner loop.
      End
   Repeat
Until Not(Check) ;* exit outer loop when Check set False
Repeat
```

Mat Statement

Assigns values to the elements of an array. Not available in expressions.

Syntax

Mat *array* = *expression*

array is a named and dimensioned array that you want to assign values to.

expression is either a single value, or the name of a dimensioned array. If *expression* is a single value, that value is assigned to all the elements of array. If it is an array, values are assigned, element by element, to array regardless of whether the dimensions of the two arrays match. Surplus values are discarded; surplus elements remain unassigned.

Remarks

You cannot use the **Mat** statement to assign values to specific elements of an array.

Examples

This example shows how to assign the same value to all elements of an array:

```
Dim MyArray(10)
Mat MyArray = "Empty"
```

This example shows how to assign the elements of one array to those of another array:

```
Dim Array1(4)
Dim Array2(2,2)
For n = 1 To 4
   Array1(n) = n         ;* Array1(1) = 1, Array1(2) = 2, and so on
Next n
Mat Array2 = Mat Array1
* Results are: Array2(1,1) = 1, Array2(1,2) = 2
*              Array2(2,1) = 3, Array2(2,2) = 4
```

MatchField Function

Searches a string and returns the part of it that matches a pattern element.

Syntax

MatchField (*string***, *pattern*, *element*)**

string is the string to be searched. If *string* does not match pattern or is a null value, an empty string is returned.

pattern is one or more pattern elements describing *string*, and can be any of the pattern codes used by the **Match** operator. If *pattern* is a null value, an empty string is returned.

element is a number, *n*, specifying that the portion of *string* that matches the *n*th element of *pattern* is returned. If *element* is a null value, it generates a runtime error.

Remarks

pattern must contain elements that describe all the characters in *string*. For example, the following statement returns an empty string because *pattern* does not cover the substring "AB" at the end of *string*:

```
MatchField ("XYZ123AB", "3X3N", 1)
```

The following statement describes the whole string and returns a value of "XYZ", which is 3X, the substring that matches the first element of the pattern:

```
MatchField ("XYZ123AB", "3X3N...", 1)
```

Examples

Q evaluates to BBB:

```
Q = MatchField("AA123BBB9","2A0N3A0N",3)
```

zip evaluates to 01234:

```
addr = '20 GREEN ST. NATICK, MA.,01234'
zip = MatchField(ADDR,"0N0X5N",3)
```

col evaluates to BLUE:

```
inv = 'PART12345 BLUE AU'
col = MatchField(INV,"10X4A3X",2)
```

In the following examples the string does not match the pattern and an empty string is returned:

```
XYZ=MatchField('ABCDE1234',"2N3A4N",1)
XYZ=
ABC=MatchField('1234AB',"4N1A",2)
ABC=
```

Mod Function

Returns the remainder after a division operation.

Syntax

Mod (*dividend***, *divisor*)**

dividend is the number to be divided. If *dividend* is a null value, null is returned.

divisor is the number to divide by. *divisor* cannot be 0. If *divisor* is a null value, null is returned.

Remarks

The **Mod** function calculates the remainder using the formula:

```
Mod (X, Y) = X - (Int (X / Y) * Y)
```

Use the **Div** function to return the result of a division operation.

Examples

The following examples show use of the **Mod** function:

```
Remainder = Mod(100, 25)      ;* result is 0
Remainder = Mod(100, 30)      ;* result is 10
```

Nap Statement

Pauses a program for the specified number of milliseconds. Not available in expressions.

Syntax

Nap [*milliseconds*]

milliseconds specifies the number of milliseconds to pause. The default value is 1. If *milliseconds* is a null value, the **Nap** statement is ignored.

Remarks

Do not use the **Nap** statement in a transform as it will slow down the IBM InfoSphere DataStage job run.

Example

This example shows **Nap** being called from within an InfoSphere DataStage before/after routine to poll for the existence of a resource, waiting for a short while between polls:

```
If NumTimesWaited < RepeatCount Then
   NumTimesWaited += 1
   Nap 500    ;* wait 500 millisecs = 1/2 a second
End
```

Neg Function

Returns the inverse of a number.

Syntax

Neg (*number***)**

number is the number you want to invert.

Example

The following example shows a use of the **Neg** function, equivalent to unary minus:

```
MyNum = 10
* Next line might be clearer than the equivalent
* construction which is: -(MyNum + 75) / 100
MyExpr = Neg(MyNum + 75) / 100
```

Not Function

Inverts the logical result of an expression.

Syntax

Not (*expression***)**

expression is the expression whose result is inverted. If *expression* is true, 0 is returned if false, 1 is returned. If *expression* is a null value, null is returned.

Remarks

expression is false if it evaluates to 0 or is an empty string. Any other value (except null) is true.

Examples

Here are some examples of the use of the **Not** function to invert the truth value of expressions:

```
Value1 = 5
Value2 = 5
Boolean = Not(Value1 - Value2);* Boolean = 1, that is, True
Boolean = Not(Value1 + Value2);* Boolean = 0, that is, False
Boolean = Not(Value1 = Value2);* Boolean = 0, that is, False
```

Null Statement

Performs no action and generates no object code.

Syntax

```
Null
```

Remarks

The **Null** statement acts as a dead end in a program. For example, you can use it with an **Else** clause if you do not want any operation to be performed when the **Else** clause is executed.

Example

The following example shows the use of the **Null** statement to make clear that a particular branch of a **Case** statement takes no action:

```
Begin Case
Case Arg1 = 'A'
* ... do something for first case.
Case Arg1 = 'B'
* ... do something for second case.
Case @True
* ... in all other cases, do nothing.
   Null
End Case
```

Num Function

Determines whether a string is numeric. If NLS is enabled, the result of this function depends on the current locale setting of the Numeric convention.

Syntax

Num (*expression***)**

expression is the expression to test. If *expression* is a number, a numeric string, or an empty string, a value of 1 is returned. If it is a null value, null is returned; otherwise 0 is returned.

Remarks

Strings that contain periods used as decimal points are considered numeric. But strings containing other characters used in formatting monetary or numeric amounts, for example, commas, dollar signs, and so on, are not considered numeric.

Examples

The following examples show the **Num** function being used to determine if a variable contains a number:

```
Arg1 = "123.45
Boolean = Num(Arg1) ;* returns 1, that is, True
Arg2 = "Section 4"
Boolean = Num(Arg2) ;* returns 0, that is, False
Arg3 = " "
Boolean = Num(Arg3) ;* False (space is not numeric)
Arg4 = ""
Boolean = Num(Arg4) ;* True (empty string is numeric)
```

Oconv Function

Converts an expression to an output format.

Syntax

Oconv (*expression***, *conversion* [@VM *conversion*] ...)**

expression is a string stored in internal format that you want to convert to an output format. If *expression* is a null value, null is returned.

conversion is one or more conversion codes specifying how the string is to be formatted. Separate multiple codes with a value mark. If *conversion* is a null value, it generates a runtime error.

Remarks

If you specify multiple codes, they are applied from left to right. The first code is applied to *expression*, then the next code is applied to the result of the first conversion, and so on.

The **Status** function uses the following values to indicate the result of an **Oconv** function:

0 The conversion was successful.

1 An invalid string was passed to the **Oconv** function. Either, the original string was returned, or if the string was a null value, null was returned.

2 The conversion was invalid.

Examples

ASCII Conversions

The following examples show the effect of some **MY** (ASCII) conversion codes.

Conversion Expression
External Value

```
X = Oconv("41424344", "MY")
       X = "ABCD"

X = Oconv("30313233", "MY")
       X = "0123"
```

Date Conversions

The following examples show the effect of various **D** (Date) conversion codes:

Conversion Expression
External Value

```
X = Oconv(0, "D")
       X = "31 DEC 1967"

X = Oconv(10740, "D2")
       X = "27 MAY 97"

X = Oconv(10740, "D2/")
       X = "05/27/97"

X = Oconv(10740, "D/E")
       X = "27/05/1997"

X = Oconv(10740, "D-YJ")
       X = "1997-147"

X = Oconv(10740, "D2*JY")
       X = "147*97"

X = Oconv(10740, "D YMD")
       X = "1997 5 27"

X = Oconv(10740, "D MY[A,2]")
       X = "MAY 97"

X = Oconv(10740, "D DMY[,A3,2]")
       X = "27 MAY 97"

X = Oconv(10740, "D/MDY[Z,Z,2]")
       X = "5/27/97"

X = Oconv(10740, "D DMY[,A,]")
       X = "27 MAY 1997"

X = Oconv(10740, "DYMD[2,2,2]")
       X = "97 05 27"

X = Oconv(10740, "DQ")
       X = "2"

X = Oconv(10740, "DMA")
       X = "MAY"

X = Oconv(10740, "DW")
       X = "2"

X = Oconv(10740, "DWA")
       X = "TUESDAY"
```

Group Conversions

The following examples show the effect of some **G** (Group) conversion codes:

Conversion Expression
External Value

```
X = Oconv("27.05.1997", "G1.2")
       X = "05.1997"

X = Oconv("27.05.1997", "G.2")
       X = "27.05"
```

Length Conversions

The following examples show the effect of some **L** (Length) conversion codes:

Conversion Expression
External Value

```
X = Oconv("QWERTYUIOP", "L0")
       X = 10

X = Oconv("QWERTYUIOP", "L7")
       X = ""

X = Oconv("QWERTYU", "L7")
       X = "QWERTYU"

X = Oconv("QWERTYUOP", "L3,5")
       X = ""

X = Oconv("QWER", "L3,5")
       X = "QWER"
```

Masked Character Conversions

The following examples show the effect of some masked character conversion codes (**MCA**, **MC/A**, **MCD**, **MCL**, **MCN**, **MC/N**, **MCP**, **MCT**, **MCU**, and **MCX**):

Conversion Expression
External Value

```
X = Oconv("John Smith 1-234", "MCA")
       X = "JohnSmith"

X = Oconv("John Smith 1-234", "MC/A")
       X = " 1-234"

X = Oconv("1234", "MCD")
       X = "4D2"

X = Oconv("1234", "MCDX")
       X = "4D2"

X = Oconv("John Smith 1-234", "MCL")
       X = "john smith 1-234"

X = Oconv("John Smith 1-234", "MCN")
       X = "1234"

X = Oconv("John Smith 1-234", "MC/N")
       X = "John Smith -"
```

```
X = Oconv("John^CSmith^X1-234", "MCP")
       X = "John.Smith.1-234"

X = Oconv("john SMITH 1-234", "MCT")
       X = "John Smith 1-234"

X = Oconv("john smith 1-234", "MCU")
       X = "JOHN SMITH 1-234"

X = Oconv("4D2", "MCX")
       X = "1234"

X = Oconv("4D2", "MCXD")
       X = "1234"
```

Masked Decimal Conversions

The following examples show the effect of some **MD** (Masked Decimal) conversion codes:

```
Conversion Expression
       External Value

X = Oconv(987654, "MD2")
       X = "9876.54"

X = Oconv(987654, "MD0")
       X = "987654"

X = Oconv(123456789, "MD2$,")
       X = "$1,234,567.89"

X = Oconv(987654, "MD24$")
       X = "$98.77"

X = Oconv(123456789, "MD2['f','.',',']")
       X = "f1.234.567,89"

X = Oconv(123456789, "MD2,['','','','SEK']")
       X = "1,234,567.89SEK"

X = Oconv(-123456789, "MD2<['#','.',',']")
       X = "#<1.234.567,89>"

X = Oconv(123456789, "MD33")
       X = "123456.789"

X = Oconv(1234567890, "MD32")
       X = "12345678.9"

X = Oconv(123456789, "MD2F")
       X = "F1234567.89"

X = Oconv(-123456789, "MD2C")
       X = "1234567.89cr"

X = Oconv(123456789, "MD2D")
       X = "1234567.89 "

X = Oconv(123456789, "MD2,D")
       X = "1,234,567.89 "

X = Oconv(1234567.89, "MD2P")
       X = "1234567.89"

X = Oconv(123, "MD3Z")
       X = ".123"
```

```
X = Oconv(987654, "MD2-Z")
       X = "9876.54"

X = Oconv(12345.678, "MD20T")
       X = "12345.67"

X = Oconv(123456, "MD2$12#")
       X = "$####1234.56"

X = Oconv(987654, "MD3,$CPZ")
       X = "$987.654 "

X = Oconv(987654, "MD2,ZP12#")
       X = "####9,876.54"
```

Masked Left and Right Conversions

The following examples show the effect of some **ML** and **MR** (Masked Left and Right) conversion codes:

```
Conversion Expression
       External Value

X = Oconv(123456789, "ML2$,")
       X = "$1,234,567.89"

X = Oconv(123, "ML3Z")
       X = ".123"

X = Oconv(123456789, "ML33")
       X = "123456.789"

X = Oconv(1234567890, "ML32")
       X = "12345678.9"

X = Oconv(-123456789, "ML2C")
       X = "1234567.89cr"

X = Oconv(123456789, "ML2D")
       X = "1234567.89db"

X = Oconv(-123456789, "ML2M")
       X = "1234567.89-"

X = Oconv(-123456789, "ML2E")
       X = "<1234567.89>"

X = Oconv(123456789, "ML2(*12)")
       X = "1234567.89**"

X = Oconv(123456789, "MR2(*12)")
       X = "**1234567.89"
```

Numeral Conversions

The following examples show the effect of some **NR** (Roman numeral) conversion codes:

```
Conversion Expression
       External Value

X = Oconv(1997, "NR")
       X = "mcmxcvii"

X = Oconv(1997000, "NR")
       X = "MCMXCVmm"
```

Pattern Matching Conversions

The following examples show the effect of some **P** (Pattern matching) conversion codes:

Conversion Expression
External Value

```
X = Oconv("123456789", "P(3N-3A-3X);(9N)")
       X = "123456789"

X = Oconv("123-ABC-A7G", "P(3N-3A-3X);(9N)")
       X = "123-ABC-A7G"

X = Oconv("ABC-123-A7G", "P(3N-3A-3X);(9N)")
       X = ""

X = Oconv("123-45-6789", "P(3N-2N-4N)")
       X = "123-45-6789"

X = Oconv("123-456-789", "P(3N-2N-4N)")
       X = ""

X = Oconv("123-45-678A", "P(3N-2N-4N)")
       X = ""
```

Radix Conversions

The following examples show the effect of some **MX**, **MO** and **MB** (Radix) conversion codes:

Conversion Expression
External Value

```
X = Oconv("1024", "MX")
       X = "400"

X = Oconv("CDE", "MX0C")
       X = "434445"

X = Oconv("1024", "MO")
       X = "2000"

X = Oconv("CDE", "MO0C")
       X = "103104105"

X = Oconv("1024", "MB")
       X = "10000000000"

X = Oconv("CDE", "MB0C")
       X = "010000110100010001000101"
```

Range Check Conversions

The following examples show the effect of the R (Range Check) conversion code:

Conversion Expression
External Value

```
X = Oconv(123, "R100,200")
       X = 123

X = Oconv(223, "R100,200")
       X = ""

X = Oconv(3.1E2, "R100,200;300,400")
       X = 3.1E2
```

Time Conversions

The following examples show the effect of some **MT** (Time) conversion codes:

Conversion Expression
External Value

```
X = Oconv(10000, "MT")
       X = "02:46"

X = Oconv(10000, "MTHS")
       X = "02:46:40am"

X = Oconv(10000, "MTH")
       X = "02:46am"

X = Oconv(10000, "MT.")
       X = "02.46"

X = Oconv(10000, "MTS")
       X = "02:46:40"
```

On...GoSub Statements

Transfer program control to an internal subroutine. Not available in expressions.

Syntax

On *index* **GoSub** *statement.label1* [, *statement.label2*] ...

On *index* specifies an expression that acts as an index to the list of statement labels. The value of *index* determines which statement label program control moves to. During execution, *index* is evaluated and rounded to an integer. If the value is 1 or less, the subroutine defined by *statement.label1* is executed. If the value is 2, the subroutine defined by *statement.label2* is executed; and so on. If the value is greater than the number of subroutines defined, the last subroutine is executed. A null value generates a runtime error.

GoSub *statement.label1*, *statement.label2* specifies a list of statement labels that program control can move to. If a statement label does not exist, it generates a compiler error.

Remarks

Use a **Return** statement in the subroutine to return program control to the statement following the **On...GoSub** statements.

The **On...GoSub** statements can be written on several lines. End each line except the last one with a comma.

Example

This example uses **On...GoSub** to call one of a set of internal subroutines within an IBM InfoSphere DataStage transform function depending on an incoming argument. The **Return** statement causes the execution to resume at the statement immediately following the **GoSub** statement. It is necessary to use a **GoTo** as shown to prevent control from accidentally flowing into the internal subroutines.

```
Function MyTransform(Arg1, Arg2)
   Reply = ""      ;* default reply
* Use particular subroutine depending on value of argument:
   On Arg2 GoSub BadValue, GoodValue1, GoodValue2, BadValue
   GoTo ExitFunction  ;* use GOTO to prevent an error
BadValue:
```

```
   Call DSTransformError("Invalid arg2 ":Arg2, MyTransform")
   Return             ;* return control following On...GoSub
GoodValue1:
   Reply = Arg1 * 99
   Return             ;* return control following On...GoSub
GoodValue2:
   Reply = Arg1 / 27
   Return             ;* return control following On...GoSub
ExitFunction:
Return(Reply)
```

On...GoTo Statement

Move program control to the specified label. Not available in expressions.

Syntax

On *index* **GoTo** *statement.label1* [, *statement.label2*] ...

On *index* specifies an expression that acts as an index to the list of statement labels. The value of *index* determines which statement label program control moves to. During execution, *index* is evaluated and rounded to an integer. If the value is 1 or less, the statement defined by *statement.label1* is executed. If the value is 2, the statement defined by *statement.label2* is executed; and so on. If the value is greater than the number of statements defined, the last statement is executed. A null value generates a runtime error.

GoTo *statement.label1*, *statement.label2* specifies a list of statement labels that program control can move to. If a statement label does not exist, it generates a compiler error.

Remarks

The **On...GoTo** statements can be written on several lines. End each line except the last one with a comma.

Example

This example uses **On...GoTo** to branch to one of a set of labels within an IBM InfoSphere DataStage transform function depending on an incoming argument:

```
Function MyTransform(Arg1, Arg2)
   Reply = ""      ;* default reply
* GoTo particular label depending on value of argument:
   On Arg2 GoTo BadValue, GoodValue1, GoodValue2, BadValue
* Note that control never returns to the next line.
BadValue:
   Call DSTransformError("Invalid arg2 ":Arg2, MyTransform")
   GoTo ExitFunction
GoodValue1:
   Reply = Arg1 * 99
   GoTo ExitFunction
GoodValue2:
   Reply = Arg1 / 27
* Drop through to end of function:
ExitFunction:
Return(Reply)
```

OpenSeq Statement

Opens a file for sequential processing. Not available in expressions.

OpenSeq

Syntax

```
OpenSeq pathname To file.variable   [On Error statements ]
   [Locked statements]
   [Then statements [Else statements]]
   [Else statements]
```

pathname is the path name of the file to be opened. If the file does not exist, the **OpenSeq** statement fails. If *pathname* is a null value, it generates a runtime error.

To *file.variable* assigns the file to *file.variable*. All statements used to process the file must refer to it using *file.variable*. If *file.variable* is a null value, it generates a fatal error.

On Error *statements* specifies statements to execute if there is a fatal error while the file is being processed. A fatal error occurs if the file cannot be opened or if *file.variable* is a null value.

Locked *statements* specifies statements to execute if the file is locked by another user. If you do not specify a **Locked** clause, and a conflicting lock exists, the program waits until the lock is released.

Then *statements* specifies the statements to execute after the file is open.

Else *statements* specifies the statements to execute if the file cannot be accessed or does not exist.

Remarks

Each sequential file reference in a BASIC program must be preceded by a separate **OpenSeq** statement for that file. **OpenSeq** sets an update record lock on the file. This prevents any other program from changing the file while you are processing it. Reset this lock using a **CloseSeq** statement after processing the file. Multiple **OpenSeq** operations on the same file only generate one update record lock so you need only include one **CloseSeq** statement per file.

If a fatal error occurs, and no **On Error** clause was specified:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.

If the **On Error** clause is taken, the value returned by the **Status** function is the error number.

Example

This is an example of opening a sequential file to check its existence:

```
OpenSeq ".\ControlFiles\File1" To PathFvar Locked
FilePresent = @True
End Then
   FilePresent = @True
   End Else
   FilePresent = @False
End
```

Pattern Matching Operators

Compares a string with a format pattern. If NLS is enabled, the result of a match operation depends on the current locale setting of the Ctype and Numeric conventions.

Syntax

```
string Match[es] pattern
```

string is the string to be compared. If *string* is a null value, the match is false and 0 is returned.

pattern is the format pattern, and can be one of the following codes:

This code...
Matches this type of string...

... Zero or more characters of any type.

0X Zero or more characters of any type.

*n***X** *n* characters of any type.

0A Zero or more alphabetic characters.

*n***A** *n* alphabetic characters.

0N Zero or more numeric characters.

*n***N** *n* numeric characters.

'*string* **'**
Exact text enclosed in double or single quotation marks.

Remarks

You can specify a negative match by preceding the code with ~ (tilde). For example, ~ 4A matches a string that does not contain four alphabetic characters. If *n* is longer than nine digits, it is used as a literal string.

If *string* matches *pattern*, the comparison returns 1, otherwise it returns 0.

You can specify multiple patterns by separating them with value marks. For example, the following expression is true if the address is either 16 alphabetic characters or 4 numeric characters followed by 12 alphabetic characters; otherwise, it is false:

```
address Matches "16A": CHAR(253): "4N12A"
```

An empty string matches the following patterns: "0A", "0X", "0N", "...", "", '', or \\.

Pwr Function

Raises the value of a number to the specified power.

Syntax

```
Pwr (number, power)
```

number is an expression evaluating to the number to be raised to *power*. If *number* is a null value, null is returned.

power specifies the power to raise *number* to. If *power* is a null value, null is returned. If *power* is not an integer, *number* must not be negative.

Remarks

On overflow or underflow, a warning is printed and 0 is returned.

Example

This is an example of the use of the **Pwr** function:

```
OppSide = Sqrt(Pwr(Side1, 2) + Pwr(Side2, 2))
```

Randomize Statement

Generates a repeatable sequence of random numbers in a specified range. Not available in expressions.

Syntax

Randomize (*expression***)**

expression evaluates to a number, *n*. The range that the random number is selected from is 0 through (*n* -1). For example, if *n* is 100, the random number is in the range 0 through 99. If no *expression* is supplied, or if *expression* is a null value, the internal time of day is used, and the sequence is different each time the program is run.

Remarks

Use the **Rnd** function instead of **Randomize** if you want to generate an unrepeatable random number sequence.

Example

This is an example of how a routine might use the **Randomize** statement to set the start seed for the **Rnd** function to generate a specific set of random numbers:

```
Randomize 1
For n = 1 To NumRecords
* Produce strings like "ID00", "ID01", "ID57", and so on
RandomId = "ID" : Fmt(Rnd(100), "R%2")
* ... do something with the generated Ids.
Next n
```

ReadSeq

Reads a line of data from a file opened for sequential processing. Not available in expressions.

Syntax

ReadSeq *variable* **From** *file.variable*[**On Error** *statements*]
{[Then *statements* [**Else** *statements*] | [**Else** *statements*]**}**

ReadSeq *variable* reads data from the current position in the file up to a newline and assigns it to *variable*.

From *file.variable* identifies the file to read. *file.variable* must have been assigned in a previous **OpenSeq** statement. If *file.variable* is a null value, or the file is not found, or the file is not open, it generates a runtime error.

On Error *statements* specifies statements to execute if there is a fatal error while the file is being processed. A fatal error occurs if the file cannot be opened or if *file.variable* is a null value.

Then *statements* specifies the statements to execute after the line is read from the file.

Else *statements* specifies the statements to execute if the file is not readable, or an end-of-file is encountered.

Remarks

The **OpenSeq** statement sets a pointer to the first line of the file. **ReadSeq** then:

1. Reads data from the current position in the file up to a newline.
   a. Assigns the data to variable.
   b. Resets the pointer to the position following the newline.
   c. Discards the newline.

If the connection between client and the computer where the engine tier resides times out, **ReadSeq** returns no bytes from the buffer, and the operation must be retried.

The **Status** function returns these values after a **ReadSeq** operation:

0 The read was successful.

1 An end-of-file was encountered.

2 The connection timed out.

-1 The file was not open.

Any other value is an error number indicating that the **On Error** clause was taken. If a fatal error occurs, and the **On Error** clause was not specified:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.

Example

The following example shows **ReadSeq** used to process each line of a sequential file:

```
OpenSeq PathName To FileVar Else
   Call DSLogWarn("Cannot open ":PathName, MyRoutine)
   GoTo ErrorExit
End
Loop
   ReadSeq FileLine From FileVar
   On Error
      Call DSLogWarn("Error from ":PathName:
      →" status=":Status(),            "MyRoutine")
      GoTo ErrorExit
   End
   Then
* ... process the line we just read
   End Else
      Exit      ;* at end-of-file
   End
Repeat
CloseSeq FileVar
```

REAL Function

Use the REAL function to convert *number* into a floating-point number without loss of accuracy. If *number* evaluates to the null value, null is returned.

Syntax

```
REAL (number)
```

Return Statement

Ends a subroutine and returns control to the calling program or statement. Not available in expressions.

Syntax

Return [**To** *statement.label*]

To *statement.label* is used with an internal subroutine initiated with **GoSub** to specify that program control returns to the specified statement label. If there is no **To** clause, control returns to the statement after the **GoSub** statement. If *statement.label* does not exist, it generates a compiler error.

Remarks

When a **Return** statement ends an external subroutine called with a **Call** statement, all files opened by the subroutine are closed, except files that are open to common variables.

Return (value) Statement

Returns a value from a user-written function. Not available in expressions.

Syntax

Return (*expression***)**

expression evaluates to the value you want the user-written function to return. If you do not specify *expression*, an empty string is returned.

Remarks

You can use the **Return (*value*)** statement only in user-written functions. If you use one in a program or subroutine, it generates an error.

Example

This example shows the use of the **Return (*value*)** statement, where the **Function** and **Deffun** statements are used to call a transform function named "MyFunctionB" from within another transform function named "MyFunctionA":

```
Function MyFunctionA(Arg1)
* When referencing a user-written function that is held in the
* DataStage repository, you must declare it as a function with
* the correct number of arguments, and add a "DSU." prefix.
Deffun MyFunctionB(A) Calling "DSU.MyFunctionB"
   Ans = MyFunctionB(Arg1)
* Add own transformation to the value in Ans...
...
Return(Ans)
```

Right Function

Extracts a substring from the end of a string.

Syntax

Right (*string***, *n*)**

string is the string containing the substring. If *string* is a null value, null is returned.

n is the number of characters to extract from the end of the string. If *n* is a null value, it generates a runtime error.

Examples

These examples extract the rightmost three characters of a string:

```
MyString = "ABCDEF"
MySubStr = Right(MyString, 3)  ;* answer is "DEF"
MySubStr = Right("AB", 3)      ;* answer is "AB"
```

Rnd Function

Generates a random number. Not available in expressions.

Syntax

Rnd (*expression***)**

expression evaluates to a number, *n*. The range that the random number is selected from is 0 through (*n* -1). For example, if *n* is 100, the random number is in the range 0 through 99. If *expression* is a negative number, a random negative number is generated. If *expression* is 0, 0 is returned. If *expression* is a null value, it causes a runtime error.

Remarks

To generate repeatable sequences of random numbers, use the **Randomize** statement instead of **Rnd**.

Example

This is an example of how a routine might use the **Randomize** statement to set the start seed for the **Rnd** function to generate a specific set of random numbers:

```
Randomize 1
For n = 1 To NumRecords
* Produce strings like "ID00", "ID01", "ID57", and so on
RandomId = "ID" : Fmt(Rnd(100), "R%2")
* ... do something with the generated Ids.
Next n
```

Seq Function

Converts an ASCII character to its numeric code value.

Syntax

Seq (*character***)**

character is the ASCII character to be converted. If *character* is a null value, null is returned.

Remarks

The **Seq** function is the inverse of the **Char** function.

Example

This example uses the **Seq** function to return the number associated with the first character in a string:

```
MyVal = Seq("A")      ;* returns 65
MyVal = Seq("a")      ;* returns 97
MyVal = Seq(" 12")    ;* returns 32 - first char is a space
MyVal = Seq("12")        ;* returns 49 - first char is digit "1"
```

SetLocale

In NLS mode, sets a locale for a specified category.

Syntax

$Include UNIVERSE.INCLUDE UVNLSLOC.H*name* = **SetLocale (***category*, *value***)**

category is one of the following include tokens:

Token Meaning

UVLC$TIME
Time and date

UVLC$NUMERIC
Numeric

UVLC$MONETARY
Currency

UVLC$CTYPE
Character type

UVLC$COLLATE
Sorting sequence

value is a locale name.

Remarks

The success of the **SetLocale** function should be tested with the **Status** function, which returns one of the following values:

Value Meaning

0 The call is successful.

LCE$NOLOCALES
NLS is not enabled for IBM InfoSphere DataStage.

LCE$BAD.LOCALE
value is not a valid locale name.

LCE$BAD.CATEGORY
The specified category is not recognized

Example

```
* Switch local time convention to Japanese
SetLocale (UVLC$TIME, "JP-JAPANESE")
If Status() <> 0 Then
   ...
End
```

Sleep Statement

Pauses a program for the specified number of seconds. Not available in expressions.

Syntax

Sleep [*seconds*]

seconds is the number of seconds to pause. If *seconds* is not specified or is a null value, a value of 1 is used.

Remarks

Do not use the **Sleep** statement in a transform as it will slow down the IBM InfoSphere DataStage job run.

Example

This example shows the **Sleep** routine being called from an InfoSphere DataStage before/after routine to poll for the existence of a resource, waiting for a short while between polls:

```
If NumTimesWaited < RepeatCount Then
   NumTimesWaited += 1
   Sleep 60                ;* 60 seconds = 1 minute
End
```

Soundex Function

Generates codes that can be used to compare character strings based on how they sound.

Syntax

Soundex (*string***)**

string is the string to be analyzed. Only the alphabetic characters in *string* are considered. If *string* is a null value, null is returned.

Remarks

The **Soundex** function returns a phonetic code consisting of the first letter of the string followed by a number. Words that sound similar, for example fare and fair, generate the same phonetic code.

Example

The following examples show the **Soundex** values for various strings:

```
MySnd = Soundex("Greenwood") ;* returns "G653"
MySnd = Soundex("Greenwod")  ;* returns "G653"
MySnd = Soundex("Green")     ;* returns "G650"
MySnd = Soundex("")          ;* returns ""
```

Space Function

Returns a string containing the specified number of blank spaces.

Syntax

Space (*spaces***)**

spaces specifies the number of spaces in the string. If *spaces* is a null value, it generates a runtime error.

Example

This is an example of the **Space** function used to generate a string with a variable number of spaces:

```
MyStr = Space(20 - Len(Arg1)):Arg1
* pad with spaces on left
```

Sqrt Function

Returns the square root of a number.

Syntax

Sqrt (*number***)**

number is 0 or a positive number. A negative number generates a runtime error. If *number* is a null value, null is returned.

Example

This is an example of the use of the **Sqrt** function:

```
OppSide = Sqrt(Side1 ^ 2 + Side2 ^ 2)
```

SQuote Function

Encloses a string in single quotation marks.

Syntax

SQuote (*string***)**

string is the string to be quoted. If *string* is a null value, an unquoted null is returned.

Example

This is an example of the **SQuote** function adding single quotation characters (') to the beginning and end of a string:

```
ProductNo = 12345
QuotedStr = SQuote(ProductNo : "A")
* result is "12345A"
```

Status Function

Returns a code that provides information about how a preceding function was executed.

Syntax

```
Status ( )
```

Remarks

The value returned by **Status** varies according to the function it is reporting. Lists of possible values are in the descriptions of the functions concerned. You can use **Status** after the following functions:

- Fmt
- Iconv
- Oconv
- OpenSeq
- ReadSeq
- WriteSeq

- WriteSeqf

Examples

Here is an example of the **Status** function being used to check the correct operation of an **Iconv** function call:

```
InDate = Iconv(ExtDate, "D2") ;* convert date to internal form
ConvStatus = Status()
Begin Case
Case ConvStatus = 0
* ...conversion succeeded
Case ConvStatus = 1
* ...conversion failed - ExtDate not parsable as a date
Case ConvStatus = 2
* ...conversion failed - conversion "D2" invalid (unlikely!)
Case ConvStatus = 3
* ...conversion succeeded, but ExtDate might have been
* invalid, for example, if it contained the string "31/02/97"
End Case
```

Here is an example of the **Status** function being used to check the correct operation of a **Fmt** function call:

```
FormattedNum = Fmt(IntNum, "R2$")   ;* format a number
FmtStatus = Status()
Begin Case
Case FmtStatus = 0
* ...formatting succeeded
Case FmtStatus = 1
* ... formatting failed - IntNum not convertable to a number
Case FmtStatus = 2
* ... formatting failed - format "R2$" invalid (unlikely!)
End Case
```

Str Function

Composes a string by repeating the input string the specified number of times.

Syntax

Str (*string*, *repeat***)**

string is the string to be repeated. If *string* is a null value, null is returned.

repeat is the number of times to repeat *string*. If *repeat* is a negative number, an empty string is returned. If *repeat* is a null value, it causes a runtime error.

Example

This is an example of the **Str** function being used to generate a string with a variable number of spaces:

```
MyStr = Str("A", 20 - Len(Arg1)):Arg1
* pad with "A"s on left
```

Subroutine Statement

Marks the start of an external subroutine. Not available in expressions.

Syntax

Subroutine [*name*] **(** *argument1*[,*argument2*]... **)**

name is a name that identifies the subroutine in any way that is helpful to make the program easy to read.

argument1 and *argument2* are the names of variables used to pass arguments between the calling program and the subroutine. A subroutine used in a transform must have one or more arguments; a before subroutine or an after subroutine must contain two arguments.

Remarks

The **Subroutine** statement must be the first noncomment line in the subroutine. Each subroutine can contain only one **Subroutine** statement. The **Call** statement that calls the subroutine must specify the same number of arguments as the **Subroutine** statement.

Example

This example shows how a before/after routine must be declared as a subroutine. The Designer client will automatically ensure this when you create a new before/after routine.

```
Subroutine MyRoutine(InputArg, ErrorCode)
* Users can enter any string value they like when using
* MyRoutine from within the job Designer. It will appear
* in the variable named InputArg.
* The routine controls the progress of the job by setting
* the value of ErrorCode, which is an Output argument.
* Anything non-zero will stop the stage or job.
ErrorCode = 0             ;* default reply
* Do some processing...
...
Return
```

Time Function

Returns the internal system time.

Syntax

```
Time ( )
```

Remarks

The internal time is taken from the computer on which the engine tier resides, and is returned as the number of seconds since midnight to the nearest thousandth of a second.

Example

This is an example of the current system wall clock time being assigned to a variable:

```
TimeInSecs = Int(Time())      ;* remove any fractional part
```

TimeDate Function

Returns the system time and date. If NLS is enabled, the result of this function depends on the current locale setting of the Time convention.

Syntax

```
TimeDate ( )
```

Remarks

The time and date are returned in the following format:

hh:mm:ss dd mmm yyyy

hh is the hours (based on a 24-hour clock).

mm is the minutes.

ss is the seconds.

dd is the day.

mmm is a three-letter abbreviation for the month.

yyyy is the year.

Example

This is an example of how a human-readable form of the current system date and time can be assigned to a variable and manipulated:

```
NowStr = TimeDate()   ;* e.g. "09:59:51 03 JUN 1997"
* extract time only
NowTimeStr = Field(NowStr, " ", 1, 1)
* extract rest as date
NowDateStr = Field(NowStr, " ", 2, 3)
```

Trigonometric Functions

The trigonometric functions return the trigonometric value specified by the function. They all have similar syntax.

General Syntax

TrigFunc (*number***)**

TrigFunc is one of the trigonometric functions: **Cos**, **Sin**, **Tan**, **ACos**, **ASin**, **ATan**, **CosH**, **TanH**, or **SinH**.

number is the number or expression you want to evaluate. If *number* is a null value, a null value is returned. If *number* is an angle, values outside the range 0 through 360 are interpreted as modulo 360. Values greater than 1E17 produce a warning message and 0 is returned.

Remarks

Cos returns the cosine of an angle. *number* is the number of degrees in the angle. **Cos** is the inverse of **ACos**.

Sin returns the sine of an angle. *number* is the number of degrees in the angle. **Sin** is the inverse of **ASin**.

Tan returns the tangent of an angle. *number* is the number of degrees in the angle. **Tan** is the inverse of **ATan**.

ACos returns the arc-cosine of *number* in degrees. **ACos** is the inverse of **Cos**.

ASin returns the arc-sine of *number* in degrees. **ASin** is the inverse of **Sin**.

ATan returns the arc-tangent of *number* in degrees. **ATan** is the inverse of **Tan**.

CosH returns the hyperbolic cosine of an angle. *number* is the number of degrees in the angle.

SinH returns the hyperbolic sine of an angle. *number* is the number of degrees in the angle.

TanH returns the hyperbolic tangent of an angle. *number* is the number of degrees in the angle.

Examples

This example shows that the **ACos** function is the inverse of the **Cos** function:

```
Angle = 45
NewAngle = Acos(Cos(Angle))  ;* NewAngle should be 45 too
```

This example shows that the **ASin** function is the inverse of the **Sin** function:

```
Angle = 45
NewAngle = Asin(Sin(Angle))  ;* NewAngle should be 45 too
```

This example shows that the **ATan** function is the inverse of the **Tan** function:

```
Angle = 45
NewAngle = Atan(Tan(Angle))  ;* NewAngle should be 45 too
```

This example uses the **Cos** function to calculate the secant of an angle:

```
Angle = 45                   ;* define angle in degrees
Secant = 1 / Cos(Angle)      ;* calculate secant
```

This example uses the **CosH** function to calculate the hyperbolic secant of an angle:

```
Angle = 45                   ;* define angle in degrees
HSecant = 1 / Cosh(Angle)    ;* calculate hyperbolic secant
```

This example uses the **Sin** function to calculate the cosecant of an angle:

```
Angle = 45                   ;* define angle in degrees
CoSecant = 1 / Sin(Angle)    ;* calculate cosecant
```

This example uses the **SinH** function to calculate the hyperbolic cosecant of an angle:

```
Angle = 45                   ;* define angle in degrees
HCoSecant = 1 / Sinh(Angle)
* calculate hyperbolic cosecant
```

This example uses the **Tan** function to calculate the cotangent of an angle:

```
Angle = 45                   ;* define angle in degrees
CoTangent = 1 / Tan(Angle)   ;* calculate cotangent
```

This example uses the **TanH** function to calculate the hyperbolic cotangent of an angle:

```
Angle = 45                   ;* define angle in degrees
HCoTangent = 1 / Tanh(Angle)
* calculate hyperbolic cotangent
```

Trim Function

Trims unwanted characters from a string.

Syntax

Trim (*string***)**
Trim (*string***, *character* [,*option*])**

string is a string containing unwanted characters. If *string* is a null value, null is returned.

character specifies a character to be trimmed (other than a space or tab). If *character* is a null value, it causes a runtime error.

option specifies the type of trim operation and can be one of the following:

L Removes leading occurrences of character.

T Removes trailing occurrences of character.

B Removes leading and trailing occurrences of character.

R Removes leading and trailing occurrences of character, and reduces multiple occurrences to a single occurrence.

A Removes all occurrences of character.

F Removes leading spaces and tabs.

E Removes trailing spaces and tabs.

D Removes leading and trailing spaces and tabs, and reduces multiple spaces and tabs to single ones.

If *option* is not specified or is a null value, **R** is assumed.

Remarks

In the first syntax, multiple occurrences of spaces and tabs are reduced to single ones, and all leading and trailing spaces and tabs are removed.

Examples

Here are some examples of the various forms of the **Trim** function:

```
MyStr = Trim("   String  with   whitespace  ")
* ...returns "String with whitespace"
MyStr = Trim("..Remove..redundant..dots....", ".")
* ...returns "Remove.redundant.dots"
MyStr = Trim("Remove..all..dots....", ".", "A")
* ...returns "Removealldots"
MyStr = Trim("Remove..trailing..dots....", ".", "T")
* ...returns "Remove..trailing..dots"
```

TrimB Function

Trims trailing spaces from a string.

Syntax

TrimB (*string***)**

string is the string that contains the trailing spaces. If *string* is a null value, null is returned.

Example

```
MyStr = TrimB("   String  with   whitespace  ")
* ...returns "("   String  with   whitespace"
```

TrimF Function

Trims leading spaces and tabs from a string.

Syntax

TrimF (*string***)**

string is the string that contains the leading spaces. If *string* is a null value, null is returned.

Example

```
MyStr = TrimF("   String  with   whitespace  ")
* ...returns "String  with   whitespace  "
```

UniChar Function

In NLS mode, generates a single character in Unicode format.

Syntax

UniChar (*expression***)**

expression is the decimal value of a Unicode character, in the range 0 to 65535.

Remarks

If *expression* has a value outside the specified range, **UniChar** returns an empty string. If *expression* is an SQL null, an SQL null is returned.

UniSeq Function

In NLS mode, converts a Unicode character to its equivalent decimal value.

Syntax

UniSeq (*expression***)**

expression is a Unicode character that is to be converted to its decimal value.

Remarks

Compare to the **Seq** function which converts ASCII characters to their decimal equivalents.

UpCase Function

Changes lowercase letters in a string to uppercase. If NLS is enabled, the result of this function depends on the current locale setting of the Ctype convention.

Syntax

UpCase (*string***)**

string is a string whose letters you want to change to uppercase.

Example

This is an example of the **UpCase** function:

```
MixedCase = "ABC123abc"
UpperCase = UpCase(MyString) ;* result is "ABC123ABC"
```

WEOFSeq Function

Writes an end-of-file mark in an open sequential file.

Syntax

WEOFSeq *file.variable* [**On Error** *statements*]

file.variable specifies the sequential file. *file.variable* is the variable name assigned to the file by the preceding **OpenSeq** statement.

On Error *statements* specify the action to take if there is a fatal error. A fatal error occurs if the file is not open, or *file.variable* is a null value. If you do not specify an **On Error** clause, the job aborts and an error is written to the job log file.

Remarks

The end-of-file mark truncates the file at the current pointer position. Any subsequent **ReadSeq** statement takes the **Else** clause.

Example

The following example opens a sequential file and truncates it by writing an end-of-file marker immediately:

```
OpenSeq PathName To FileVar Then
   WeofSeq FileVar
End Else
   Call DSLogFatal("Cannot open file ":Pathname,"Routine1")
   GoTo ErrorExit
End
```

WriteSeq Function

Writes a new line to a file that is open for sequential processing and advances a pointer to the next position in the file.

Syntax

WriteSeq *line* **To** *file.variable*
[**On Error** *statements*]
{[**Then** *statements* [**Else** *statements*] | [**Else** *statements*]**}**

line is the line to write to the sequential file. **WriteSeq** writes a newline at the end of the line.

To *file.variable* specifies the sequential file. *file.variable* is the variable name assigned to the file by the preceding **OpenSeq** statement.

On Error *statements* specify the action to take if there is a fatal error. A fatal error occurs if the file is not open, or *file.variable* is a null value. If you do not specify an **On Error** clause, the job aborts and an error message is written to the job log file.

Then *statements* specify the action the program takes after the line is written to the file. If you do not specify a **Then** clause, you must specify an **Else** clause.

Else *statements* specify the action the program takes if the line cannot be written to the file, for example, if the file does not exist. If you do not specify an **Else** clause, you must specify a **Then** clause.

Remarks

The line is written at the current position in the file and then the pointer is advanced to the next position after the newline. Any existing data in the file is overwritten, unless the pointer is at the end of the file.

You can use the **Status** function after **WriteSeq** to determine the success of the operation. **Status** returns 0, if the file was locked, -2 if the file was not locked, and an error code if the **On Error** clause was taken.

Example

The following example writes a single line to a sequential file by truncating and then writing to it immediately after it is opened:

```
OpenSeq PathName To FileVar Then
   WeofSeq FileVar        ;* write end-of-file mark immediately
   WriteSeq "First line" To FileVar Else
   On Error
      Call DSLogWarn("Error from ":PathName:"
        → status=":Status(),                  "MyRoutine")
      GoTo ErrorExit
   End
      Call DSLogFatal("Cannot write to ":Pathname,
        →   "MyRoutine")
      GoTo ErrorExit
   End
End Else
   Call DSLogFatal("Cannot open file ":Pathname, "MyRoutine")
   GoTo ErrorExit
End
```

WriteSeqF Function

Writes a new line to a file that is open for sequential processing, advances a pointer to the next position in the file, and saves the file to disk.

Syntax

```
WriteSeqF line To file.variable
[On Error statements]
{[Then statements [Else statements]] | [Else statements]}
```

line is the line to write to the sequential file. **WriteSeqF** writes a newline at the end of the line.

To *file.variable* specifies the sequential file. *file.variable* is the variable name assigned to the file by the preceding **OpenSeq** statement.

On Error *statements* specify the action to take if there is a fatal error. A fatal error occurs if the file is not open, or *file.variable* is a null value. If you do not specify an **On Error** clause, the job aborts and an error message is written to the job log file.

Then *statements* specify the action the program takes after the line is written to the file. If you do not specify a **Then** clause, you must specify an **Else** clause.

Else *statements* specify the action the program takes if the line cannot be written to the file, for example, if the file does not exist. If you do not specify an **Else** clause, you must specify a **Then** clause.

Remarks

WriteSeqF works in the same way as **WriteSeq**, except that each line is written directly to disk instead of being buffered and then being written in batches. A **WriteSeqF** statement after several **WriteSeq** statements writes all buffered lines to disk.

Note: Use the **WriteSeqF** statement for logging operations only as the increased disk I/O slows down program performance.

You can use the **Status** function after **WriteSeqF** to determine the success of the operation. **Status** returns 0, if the file was locked, -2 if the file was not locked, and an error code if the **On Error** clause was taken.

Example

The following example appends to a sequential file by reading to the end of it, then force-writing a further line:

```
OpenSeq PathName To FileVar Then
   Loop
      ReadSeq Dummy From FileVar Else Exit ;* at end-of-file
   Repeat
   WriteSeqF "Extra line" To FileVar Else
   On Error
      Call DSLogWarn("Error from ":PathName:"
      → status=":Status(),                  "MyRoutine")
      GoTo ErrorExit
   End
   Call DSLogFatal("Cannot write to ":Pathname, "MyRoutine")
      GoTo    ErrorExit
   End
End Else
   Call DSLogFatal("Cannot open file ":Pathname, "MyRoutine")
   GoTo ErrorExit
End
```

Xtd Function

Converts a hexadecimal string to decimal.

Syntax

Xtd (*string***)**

string is the numeric string you want to convert.

Example

This is an example of the **Xtd** function used to convert a decimal number to a hexadecimal string representation:

```
MyHex = "2F"
MyNumber = Xtd(MyHex)     ;* returns 47
```

Conversion Codes

Conversion codes specify how data is formatted for output or internal storage. They are specified in an **Iconv** or **Oconv** function. Here is a list of the codes you can use.

Extracting characters from fields:

G Extracting field values **MCA** Extracting alphabetic characters from a field **MC/A** Extracting nonalphabetic characters from a field **MCN** Extracting numeric characters from a field **MC/N** Extracting nonnumeric characters from a field **MCM** Extracting NLS multibyte characters from a field **MC/M** Extracting NLS single-byte characters from a field **P** Extracting data that matches a pattern **R** Extracting a numeric value that falls within a range

Preprocessing data:

L Limiting the length of returned data **S** Generating codes to compare words by how they sound

Processing text:

MCU Converting lowercase letters to uppercase **MCL** Converting uppercase letters to lowercase **MCT** Converting words in the field to initial capitals **MCP** Converting unprintable characters to a period **NLS** Converting strings between internal and external format using a character set map

Formatting numbers, dates, times, and currency:

MD Formatting numbers as monetary or numeric amounts **ML** Left-justifying and formatting numbers **MR** Right-justifying and formatting numbers **MP** Packing decimal numbers two-per-byte for storage **D** Converting dates **MT** Converting times **TI** Converting times in internal format to default local convention **NR** Converting Roman numerals into Arabic numerals **NL** Converting locale-dependent alternative characters to Arabic numerals **MM** Formatting currency data

Radix conversions:

MX Converting hexadecimal numbers to decimal **MCD** Converting decimal numbers to hexadecimal **MCX** Converting hexadecimal numbers to decimal **MO** Converting octal numbers to decimal **MB** Converting binary numbers to decimal **MY** Converting hexadecimal numbers to their ASCII equivalents **MUOC** Converting hexadecimal numbers to Unicode character values

The conversion codes are described in more detail in the following reference pages. The conversion codes appear in alphabetical order.

D

Converts dates to storage format and vice versa. When NLS is enabled, the locale default date format overrides any default date format set in the msg.text file.

Syntax

```
D [years.digits] [delimiter skip] [separator] [format.options
[ modifiers ] ] [ E ] [ L ]
```

years.digits indicates the number of digits of the year to output. The default is 4. On input *years.digits* is ignored. If the input date has no year, the year is taken from the system date.

delimiter is any single nonnumeric character used as a field delimiter in the case where conversion must first do a group extraction to obtain the internal date. It cannot be the system delimiter.

skip must accompany the use of *delimiter* and is the number of delimited fields to skip in order to extract the date.

separator is the character used to separate the day, month, and year on output. If you do not specify *separator*, the date is converted in the form 01 DEC 1999. On input *separator* is ignored. If NLS is enabled and you do not specify *years.digits* or *separator*, the default date form is 01 DEC 1999.

format.options is up to six options that define how the date is output (they are ignored on input). Each format option can have an associated modifier, described below. Format options can only be used in certain combinations as described below. The options are as follows:

- **Y** [*n*] outputs the year number as *n* digits.
- **YA** outputs the name of the Chinese calendar year only. If NLS is enabled, uses the YEARS field in the Time/Date locale.
- **M** outputs the month only as a number from 1 through 12.
- **MA** outputs only the month's name. If NLS is enabled, uses the MONS field in the Time/Date locale. You can use any combination of uppercase and lowercase letters for the month; IBM InfoSphere DataStage checks the combination against the ABMONS field, otherwise the MONS field.
- **MB** outputs the abbreviated month name. If NLS is enabled, uses the ABMONS field in the Time/Date locale; otherwise, uses the first three characters of the month name.
- **MR** outputs the month number in Roman numerals.
- **D** outputs the day of the month as a number from 1 through 31.
- **W** outputs the day of the week as a number from 1 through 7, where Monday is 1. If NLS is enabled, uses the DAYS field in the Time/Date locale, where Sunday is 1.
- **WA** outputs the day by name. If NLS is enabled, uses the DAYS field in the Time/Date locale, unless modified by the format modifiers, *f1*, *f2*, and so forth.
- **WB** outputs the abbreviated day name. If NLS is enabled, uses the ABDAYS field in the Time/Date locale.
- **Q** outputs the quarter of the year as a number from 1 through 4.
- **J** outputs the day of the year as a number, 1 through 366.
- **N** outputs the year number within the current era. If NLS is enabled, uses the ERA STARTS field in the Time/Date locale.
- **NA** outputs the era name corresponding to the current year. If NLS is enabled, uses the ERA NAMES or ERA STARTS fields in the Time/Date locale.
- **Z** outputs the time zone name.

The following shows which format options can be used together:

Use this option...	**With these options...**
Y	M, MA, D, J, [*modifiers*]
YA	M, MA, D, [*modifiers*]
M	Y, YA, D, [*modifiers*]
MA	Y, YA, D, [*modifiers*]
MB	Y, YA, D, [*modifiers*]
D	Y, M, [*modifiers*]
N	Y, M, MA, MB, D, WA, [*modifiers*]
NA	Y, M, MA, MB, D, WA, [*modifiers*]
W	Y, YA, M, MA, D
WA	Y, YA, M, MA, D
WB	Y, YA, M, MA, D

Q

J Y, [*modifiers*]

Z [*modifiers*]

[*modifiers*] modify the output formats for the data specified by *format.options*. You can specify up to six modifiers, separated by commas. The commas indicate which *format.option* each modifier is associated with, therefore you must include all the commas, even if you want to specify only one modifier (see examples). They can be any of the following values:

- *n* displays *n* characters. It is used with the D, M, Y, W, Q and J numeric options. It is used with MA, MB, WA, WB, YA, N, "*text*" text options.
- A[*n*] displays the month as *n* alphabetic characters. It is used with the Y, M, W, and N options.
- Z[*n*] suppresses leading zeros and displays as *n* digits. It works as n with numeric options.
- E toggles day/month/year and month/day/year format for dates.
- L displays month or day names as lowercase. The default is uppercase.

Value Returned by the Status Function

If you input an invalid date to this code, it returns a valid internal date but flags the anomaly by assigning a **Status** function value of 3. For example, 02/29/99 is interpreted as 03/01/99, and 09/31/93 is interpreted as 10/01/93. If the input date is a null value, **Status** is assigned a value of 3 and no conversion occurs.

Examples

The following examples show the effect of various **D** conversion codes with the **Iconv** function:

Conversion Expression
Internal Value

```
X = Iconv("31 DEC 1967", "D")
       X = 0

X = Iconv("27 MAY 97", "D2")
       X = 10740

Iconv("05/27/97", "D2/")
       X = 10740

X = Iconv("27/05/1997", "D/E")
       X = 10740

X = Iconv("1997 5 27", "D YMD")
       X = 10740

X = Iconv("27 MAY 97", "D DMY[,A3,2]")
       X = 10740

X = Iconv("5/27/97", "D/MDY[Z,Z,2]")
       X = 10740

X = Iconv("27 MAY 1997", "D DMY[,A,]")
       X = 10740

X = Iconv("97 05 27", "DYMD[2,2,2]")
       X = 10740
```

The following examples show the effect of various **D** conversion codes with the **Oconv** function:

Conversion Expression
External Value

X = Oconv(0, "D")
X = "31 DEC 1967"

X = Oconv(10740, "D2")
X = "27 MAY 97"

X = Oconv(10740, "D2/")
X = "05/27/97"

X = Oconv(10740, "D/E")
X = "27/05/1997"

X = Oconv(10740, "D-YJ")
X = "1997-147"

X = Oconv(10740, "D2*JY")
X = "147*97"

X = Oconv(10740, "D YMD")
X = "1997 5 27"

X = Oconv(10740, "D MY[A,2]")
X = "MAY 97"

X = Oconv(10740, "D DMY[,A3,2]")
X = "27 MAY 97"

X = Oconv(10740, "D/MDY[Z,Z,2]")
X = "5/27/97"

X = Oconv(10740, "D DMY[,A,]")
X = "27 MAY 1997"

X = Oconv(10740, "DYMD[2,2,2]")
X = "97 05 27"

X = Oconv(10740, "DQ")
X = "2"

X = Oconv(10740, "DMA")
X = "MAY"

X = Oconv(10740, "DW")
X = "2"

X = Oconv(10740, "DWA")
X = "TUESDAY"

G

Extracts one or more delimited values from a field.

G

Syntax

G [*skip*] *delimiter fields*

skip specifies the number of fields to skip; if it is not specified, 0 is assumed and no fields are skipped.

delimiter is a nonnumeric character used as the field separator. You must not use the system variables @IM, @FM, @VM, @ SM, and @TM as delimiters.

fields is the number of contiguous values to extract.

Examples

The following examples show the effect of some **G** conversion codes with the **Iconv** function:

Conversion Expression
Internal Value

```
X = Iconv("27.05.1997", "G1.2")
      X = "05.1997"

X = Iconv("27.05.1997", "G.2")
      X = "27.05"
```

The following examples show the effect of some **G** conversion codes with the **Oconv** function:

Conversion Expression
External Value

```
X = Oconv("27.05.1997", "G1.2")
      X = "05.1997"

X = Oconv("27.05.1997", "G.2")
      X = "27.05"
```

L

Extracts data that meets length criteria.

Syntax

```
L [ n [ ,m ] ]
```

n on its own is the maximum number of characters that the data must contain in order to be returned. If it contains more than *n* characters, an empty string is returned. If you do not specify *n*, or if *n* is 0, the length of the data is returned.

n, *m* specifies a range. If the data contains *n* through *m* characters it is returned, otherwise an empty string is returned.

Examples

The following examples show the effect of some **L** conversion codes with the **Iconv** function:

Conversion Expression
Internal Value

```
X = Iconv("QWERTYUIOP", "L0")
      X = 10

X = Iconv("QWERTYUIOP", "L7")
      X = ""

X = Iconv("QWERTYU", "L7")
      X = "QWERTYU"

X = Iconv("QWERTYUOP", "L3,5")
      X = ""
```

```
X = Iconv("QWER", "L3,5")
       X = "QWER"
```

The following examples show the effect of some **L** conversion codes with the **Oconv** function:

```
Conversion Expression
       External Value
X = Oconv("QWERTYUIOP", "L0")
       X = 10
X = Oconv("QWERTYUIOP", "L7")
       X = ""
X = Oconv("QWERTYU", "L7")
       X = "QWERTYU"
X = Oconv("QWERTYUOP", "L3,5")
       X = ""
X = Oconv("QWER", "L3,5")
       X = "QWER"
```

MB

Converts binary numbers to decimal or an ASCII value, for storage, or vice versa, for output.

Syntax

```
MB [ 0C ]
```

0C converts the octal number to its equivalent ASCII character on input, and vice versa on output.

Remarks

Characters other than 0 and 1 cause an error.

Examples

The following examples show the effect of some **MB** conversion codes with the **Iconv** function:

```
Conversion Expression
       Internal Value
X = Iconv("10000000000", "MB")
       X = 1024
X = Iconv("010000110100010001000101", "MB0C")
       X = "CDE"
```

The following examples show the effect of some **MB** conversion codes with the **Oconv** function:

```
Conversion Expression
       External Value
X = Oconv("1024", "MB")
       X = "10000000000"
X = Oconv("CDE", "MB0C")
       X = "010000110100010001000101"
```

MCA

Extracts all alphabetic characters in a field.

Syntax

```
MCA
```

Examples

The following example shows the effect of an **MCA** conversion code with the **Iconv** function:

Conversion Expression
Internal Value

```
X = Iconv("John Smith 1-234", "MCA")
```
X = "JohnSmith"

The following example shows the effect of an **MCA** conversion code with the **Oconv** function:

Conversion Expression
External Value

```
X = Oconv("John Smith 1-234", "MCA")
```
X = "JohnSmith"

MC/A

Extracts all nonalphabetic characters in a field.

Syntax

```
MC/A
```

Examples

The following example shows the effect of an **MC/A** conversion code with the **Iconv** function:

Conversion Expression
Internal Value

```
X = Iconv("John Smith 1-234", "MC/A")
```
X = " 1-234"

The following example shows the effect of an **MC/A** conversion code with the **Oconv** function:

Conversion Expression
External Value

```
X = Oconv("John Smith 1-234", "MC/A")
```
X = " 1-234"

MCD

Converts decimal numbers to hexadecimal.

Syntax

```
MCD
```

Examples

The following example shows the effect of an **MCD** conversion code with the **Iconv** function:

Conversion Expression
Internal Value

```
X = Iconv("4D2", "MCD")
       X = "1234"
```

The following example shows the effect of an **MCD** conversion code with the **Oconv** function:

Conversion Expression
External Value

```
X = Oconv("1234", "MCD")
       X = "4D2"
```

MCL

Converts all uppercase letters to lowercase.

Syntax

```
MCL
```

Examples

The following example shows the effect of an **MCL** conversion code with the **Iconv** function:

Conversion Expression
Internal Value

```
X = Iconv("John Smith 1-234", "MCL")
       X = "john smith 1-234"
```

The following example shows the effect of an **MCL** conversion code with the **Oconv** function:

Conversion Expression
External Value

```
X = Oconv("John Smith 1-234", "MCL")
       X = "john smith 1-234"
```

MCM

For use if NLS is enabled. Extracts all NLS multibyte characters in the field. If NLS mode is disabled, the code returns a value of 2, which indicates an invalid conversion code.

Syntax

```
MCM
```

Example

The following example shows the effect of an **MCM** conversion code with the **Iconv** function:

```
IF SYSTEM(NL$ON)
THEN
   Multibyte.Characters = ICONV(Input.String, "MCM")
END
```

Oconv behaves the same way as **Iconv**.

MC/M

For use if NLS is enabled. Extracts all single-byte characters in the field. If NLS mode is disabled, the code returns a value of 2, which indicates an invalid conversion code.

MC/M

Syntax

```
MC/M
```

Example

The following example shows the effect of an **MC/M** conversion code with the **Iconv** function:

```
IF SYSTEM(NL$ON)
THEN
   Singlebyte.Characters = ICONV(Input.String, "MC/M")
END
```

Oconv behaves the same way as **Iconv**.

MCN

Extracts all numeric characters in a field.

Syntax

```
MCN
```

Examples

The following example shows the effect of an **MCN** conversion code with the **Iconv** function:

Conversion Expression
Internal Value

X = Iconv("John Smith 1-234", "MCN")
X = "1234"

The following example shows the effect of an **MCN** conversion code with the **Oconv** function:

Conversion Expression
External Value

X = Oconv("John Smith 1-234", "MCN")
X = "1234"

MC/N

Extracts all nonnumeric characters in a field.

Syntax

```
MC/N
```

Examples

The following example shows the effect of an **MC/N** conversion code with the **Iconv** function:

Conversion Expression
Internal Value

X = Iconv("John Smith 1-234", "MC/N")
X = "John Smith -"

The following example shows the effect of an **MC/N** conversion code with the **Oconv** function:

Conversion Expression
External Value

X = Oconv("John Smith 1-234", "MC/N")
X = "John Smith -"

MCP

Converts unprintable characters to a period.

Syntax

MCP

Examples

The following example shows the effect of an **MCP** conversion code with the **Iconv** function:

Conversion Expression
Internal Value

X = Iconv("John^CSmith^X1-234", "MCP")
X = "John.Smith.1-234"

The following example shows the effect of an **MCP** conversion code with the **Oconv** function:

Conversion Expression
External Value

X = Oconv("John^CSmith^X1-234", "MCP")
X = "John.Smith.1-234"

MCT

Converts words in a string to initial capitals.

Syntax

MCT

Examples

The following example shows the effect of an **MCT** conversion code with the **Iconv** function:

Conversion Expression
Internal Value

X = Iconv("john SMITH 1-234", "MCT")
X = "John Smith 1-234"

The following example shows the effect of an **MCT** conversion code with the **Oconv** function:

Conversion Expression
External Value

```
X = Oconv("john SMITH 1-234", "MCT")
```
X = "John Smith 1-234"

MCU

Converts all lowercase letters to uppercase.

Syntax

```
MCU
```

Examples

The following example shows the effect of an **MCU** conversion code with the **Iconv** function:

Conversion Expression
Internal Value

```
X = Iconv("john smith 1-234", "MCU")
```
X = "JOHN SMITH 1-234"

The following example shows the effect of an **MCU** conversion code with the **Oconv** function:

Conversion Expression
External Value

```
X = Oconv("john smith 1-234", "MCU")
```
X = "JOHN SMITH 1-234"

MCX

Converts hexadecimal numbers to decimal.

Syntax

MCX

Examples

The following example shows the effect of an **MCX** conversion code with the **Iconv** function:

Conversion Expression
Internal Value

```
X = Iconv("1234", "MCX")
```
X = "4D2"

The following example shows the effect of an **MCX** conversion code with the **Oconv** function:

Conversion Expression
External Value

```
X = Oconv("4D2", "MCX")
```
X = "1234"

MD

Formats numbers as monetary or numeric amounts, or converts formatted numbers to internal storage format. If the **$**, **F**, **I**, or **Y** options are included, the conversion is monetary.

If NLS is enabled and the conversion is monetary, the thousands separator and decimal separator are taken from the locale MONETARY convention. If the conversion is numeric, they are taken from the NUMERIC convention. The **<**, **-**, **C**, and **D** options define numbers intended for monetary use, and override settings in the MONETARY convention.

Syntax

```
MD n [ m ] [ options ]
```

n is a number, 0 through 9, that indicates the number of decimal places used in the output. If *n* is 0, the output contains no decimal point.

m specifies the scaling factor. On input, the decimal point is moved *m* places to the right before storing. On output, the decimal point is moved *m* places to the left. For example, if *m* is 2 in an input conversion and the input data is 123, it would be stored as 12300. If *m* is 2 in an output conversion and the stored data is 123, it would be output as 1.23. If *m* is not specified, it is assumed to be the same as *n*. Numbers are rounded or padded with zeros as required.

options are any of the following:

- **,** specifies a comma as the thousands delimiter. To specify a different character as the thousands delimiter, use the convention expression.
- **$** prefixes a local currency sign to the number. If NLS is enabled, the sign is derived from the locale MONETARY convention.
- **F** prefixes a franc sign to the number.
- **I** is used with **Oconv**, specifies that the international monetary symbol for the locale is used. Used with **Iconv**, specifies that it is removed.
- **Y** is used with **Oconv**. The yen/yuan character is used.
- **-** specifies a minus sign as a suffix for negative amounts; positive amounts are suffixed with a blank space.
- **<** specifies that negative amounts are enclosed in angle brackets for output; positive amounts are prefixed and suffixed with a blank space.
- **C** adds the suffix CR to negative amounts; positive amounts are suffixed with two blank spaces.
- **D** adds the suffix DB to negative amounts; positive amounts are suffixed with two blank spaces.
- **P** specifies no scaling if input data already contains a decimal point.
- **Z** outputs 0 as an empty string.
- **T** truncates, rather than rounds, the data.
- *fx* adds a format mask on output and removes it on input. *f* is a number, 1 through 99 indicating the maximum number of mask characters to remove or add. *x* is the character used as the format mask. If you do not use the *fx* option and the data contains a format mask, an empty string results. Format masks are described in Format Expression.
- *intl* is an expression used to specify a convention for monetary or numeric formatting.
- *convention* is an expression used to specify a convention for monetary or numeric formatting.

  The *convention* expression has the following syntax:

  [*prefix*, *thousands*, *decimal*, *suffix*]

  Note: Each element of the convention expression is optional, but you must specify the brackets and the commas in the right position. For example, to specify thousands only, enter `[,thousands,, ]`.

– *prefix* specifies a prefix for the number. If *prefix* contains spaces, commas, or right square brackets, enclose it in quotation marks.
– *thousands* specifies the thousands delimiter.
– *decimal* specifies the decimal delimiter.
– *suffix* specifies a suffix for the number. If *suffix* contains spaces, commas, or right square brackets, enclose it in quotation marks.

Examples

The following examples show the effect of some **MD** (masked decimal) conversion codes with the **Iconv** function:

Conversion Expression
Internal Value

```
X = Iconv("9876.54", "MD2")
       X = 987654

X = Iconv("987654", "MD0")
       X = 987654

X = Iconv("$1,234,567.89", "MD2$,")
       X = 123456789

X = Iconv("123456.789", "MD33")
       X = 123456789

X = Iconv("12345678.9", "MD32")
       X = 1234567890

X = Iconv("F1234567.89", "MD2F")
       X = 123456789

X = Iconv("1234567.89cr", "MD2C")
       X = -123456789

X = Iconv("1234567.89 ", "MD2D")
       X = 123456789

X = Iconv("1,234,567.89 ", "MD2,D")
       X = 123456789

X = Iconv("9876.54", "MD2-Z")
       X = 987654

X = Iconv("$####1234.56", "MD2$12#")
       X = 123456

X = Iconv("$987.654 ", "MD3,$CPZ")
       X = 987654

X = Iconv("####9,876.54", "MD2,ZP12#")
       X = 987654
```

The following examples show the effect of some **MD** (Masked Decimal) conversion codes with the **Oconv** function:

Conversion Expression
External Value

```
X = Oconv(987654, "MD2")
       X = "9876.54"
```

```
X = Oconv(987654, "MD0")
       X = "987654"

X = Oconv(123456789, "MD2$,")
       X = "$1,234,567.89"

X = Oconv(987654, "MD24$")
       X = "$98.77"

X = Oconv(123456789, "MD2['f','.',',']")
       X = "f1.234.567,89"

X = Oconv(123456789, "MD2,['','','','SEK']"
       X ="1,234,567.89SEK"

X = Oconv(-123456789, "MD2<['#','.',',']")
       X = #<1.234.567,89>"

X = Oconv(123456789, "MD33")
       X = "123456.789"

X = Oconv(1234567890, "MD32")
       X = "12345678.9"

X = Oconv(123456789, "MD2F")
       X = "F1234567.89"

X = Oconv(-123456789, "MD2C")
       X = "1234567.89cr"

X = Oconv(123456789, "MD2D")
       X = "1234567.89 "

X = Oconv(123456789, "MD2,D")
       X = "1,234,567.89 "

X = Oconv(1234567.89, "MD2P")
       X = "1234567.89"

X = Oconv(123, "MD3Z")
       X = ".123"

X = Oconv(987654, "MD2-Z")
       X = "9876.54"

X = Oconv(12345.678, "MD20T")
       X = "12345.67"

X = Oconv(123456, "MD2$12#")
       X = "$####1234.56"

X = Oconv(987654, "MD3,$CPZ")
       X = "$987.654 "

X = Oconv(987654, "MD2,ZP12#")
       X = "####9,876.54"
```

ML & MR

Justifies and formats monetary or numeric amounts. **ML** specifies left justification, **MR** specifies right justification. If the **F** or **$** options are included, the conversion is monetary.

If NLS is enabled and the conversion is monetary, the thousands separator and decimal separator are taken from the locale MONETARY convention. If the conversion is numeric, they are taken from the NUMERIC convention. The **<**, **-**, **C** and **D** options define numbers intended for monetary use, and override settings in the MONETARY convention.

ML and MR

Syntax

ML | **MR** [*n* [*m*] *options* [(*fx*)]]

n is a number, 0 through 9, that indicates the number of decimal places used in the output. If *n* is 0, the output contains no decimal point.

m specifies the scaling factor. On input, the decimal point is moved *m* places to the right before storing. On output, the decimal point is moved *m* places to the left. For example, if *m* is 2 in an input conversion and the input data is 123, it would be stored as 12300. If *m* is 2 in an output conversion and the stored data is 123, it would be output as 1.23. If *m* is not specified, it is assumed to be the same as *n*. Numbers are rounded or padded with zeros as required.

options are any of the following:

- **,** specifies a comma as the thousands delimiter. To specify a different character as the thousands delimiter, use the convention expression.
- **C** adds the suffix CR to negative amounts; positive amounts are suffixed with two blank spaces.
- **D** adds the suffix DB to negative amounts; positive amounts are suffixed with two blank spaces.
- **Z** outputs 0 as an empty string.
- **M** specifies a minus sign as a suffix for negative amounts. Positive amounts are suffixed with a blank space.
- **E** specifies that negative amounts are enclosed in angle brackets for output; positive amounts are prefixed and suffixed with a blank space.
- **N** suppresses the minus sign on negative numbers.
- **$** prefixes a local currency sign to the number before justification. If NLS is enabled, the sign is derived from the locale MONETARY convention. To prefix a different monetary symbol, use the *intl* expression.
- **F** prefixes a franc sign to the number.
- **(f***x* **)** adds a format mask on output and removes it on input. *x* is a number, 1 through 99 indicating the maximum number of mask characters to remove or add. **f** is a code specifying the character used as the format mask, and is one of the following:
  - **#** specifies a mask of blanks.
  - ***** specifies a mask of asterisks.
  - **%** specifies a mask of zeros.
- *intl* is an expression used to customize output according to different international conventions, allowing multibyte characters.

  The *intl* expression has the following syntax:

  [*prefix* , *thousands* , *decimal* , *suffix*]

  Note: Each element of the convention expression is optional, but you must specify the brackets and the commas in the right position. For example, to specify thousands only, enter `[,thousands,,]`.
  - *prefix* specifies a prefix for the number. If *prefix* contains spaces, commas, or right square brackets, enclose it in quotation marks.
  - *thousands* specifies the thousands delimiter. If *thousands* contains spaces, commas, or right square brackets, enclose it in quotation marks.

– *decimal* specifies the decimal delimiter. If *decimal* contains spaces, commas, or right square brackets, enclose it in quotation marks.
– *suffix* specifies a suffix for the number. If *suffix* contains spaces, commas, or right square brackets, enclose it in quotation marks.

Literal strings can also be enclosed in parenthesis. Format masks are described in Format Expression.

Examples

The following examples show the effect of some **ML** and **MR** conversion codes with the **Iconv** convention:

Conversion Expression
Internal Value

```
X = Iconv("$1,234,567.89", "ML2$,")
      X = 123456789

X = Iconv(".123", "ML3Z")
      X = 123

X = Iconv("123456.789", "ML33")
      X = 123456789

X = Iconv("12345678.9", "ML32")
      X = 1234567890

X = Iconv("1234567.89cr", "ML2C")
      X = -123456789

X = Iconv("1234567.89db", "ML2D")
      X = 123456789

X = Iconv("1234567.89-", "ML2M")
      X = -123456789

X = Iconv("<1234567.89>", "ML2E")
      X = -123456789

X = Iconv("1234567.89**", "ML2(*12)")
      X = 123456789

X = Iconv("**1234567.89", "MR2(*12)")
      X = 123456789
```

The following examples show the effect of some **ML** and **MR** conversion codes with the **Oconv** function:

Conversion Expression
External Value

```
X = Oconv(123456789, "ML2$,")
      X = "$1,234,567.89"

X = Oconv(123, "ML3Z")
      X = ".123"

X = Oconv(123456789, "ML33")
      X = "123456.789"

X = Oconv(1234567890, "ML32")
      X = "12345678.9"

X = Oconv(-123456789, "ML2C")
      X = 1234567.89cr"
```

```
X = Oconv(123456789, "ML2D")
       X = " "1234567.89db"
X = Oconv(-123456789, "ML2M")
       X = "1234567.89-"
X = Oconv(-123456789, "ML2E")
       X = "<1234567.89>"
X = Oconv(123456789, "ML2(*12)")
       X = "1234567.89**"
X = Oconv(123456789, "MR2(*12)")
       X = "**1234567.89"
```

MM

In NLS mode, formats currency data using the current MONETARY convention.

MM

Syntax

MM [*n*] [**I** [**L**]]

n is the number of decimal places to be output or stored.

I formats the data using the three-character international currency symbol specified in the MONETARY convention for the current locale, a period for the decimal separator, and a comma for the thousands separator.

Adding **L** formats the data using the thousands separator and decimal separator in the MONETARY convention of the current locale. Both **I** and **L** are ignored for input conversions using **Iconv**.

Remarks

If you specify **MM** with no arguments, the conversion uses the decimal and thousands separators and the currency symbol specified in the MONETARY convention of the current locale.

MO

Converts octal numbers to decimal, or an ASCII value for storage, or vice versa, for output.

Syntax

MO [**0C**]

0C converts the octal number to its equivalent ASCII character on input, and vice versa on output.

Remarks

Characters outside of the range 0 through 7 cause an error.

Examples

The following examples show the effect of some **M0** conversion codes with the **Iconv** function:

Conversion Expression
Internal Value

```
X = Iconv("2000", "MO")
       X = 1024

X = Iconv("103104105", "MO0C")
       X = "CDE"
```

The following examples show the effect of some **M0** conversion codes with the **Oconv** function:

Conversion Expression
External Value

```
X = Oconv("1024", "MO")
       X = "2000"

X = Oconv("CDE", "MO0C")
       X = "103104105"
```

MP

Packs decimal numbers two per byte for storage and unpacks them for output.

Syntax

```
MP
```

Remarks

Leading + signs are ignored. Leading - signs cause a hexadecimal D to be stored in the lower half of the last internal digit. If there is an odd number of packed halves, four leading bits of 0 are added. The range of the data bytes in internal format expressed in hexadecimal is 00 through 99 and 0D through 9D.

This conversion only accepts decimal digits, 0 through 9, and plus and minus signs as input, otherwise the conversion fails.

Packed decimal numbers must be unpacked for output or they cannot be displayed.

MT

Converts data to and from time format.

MT

Syntax

MT [**H**] [**S**] [*separator*]

MT with no options specifies that time is in 24-hour format, omitting seconds, with a colon used to separate hours and minutes, for example: 23:59.

H specifies an output format in 12-hour format with the suffix AM or PM.

S includes seconds in the output time.

separator is a nonnumeric character that specifies the separator used between hours, minutes, and seconds in the output.

Remarks

On output, **MT** defines the external output format for the time.

On input, **MT** specifies only that the data is a time, and the **H** and **S** options are ignored. If the input date has no minutes or seconds, they are assumed to be 0. For 12-hour formats, use a suffix of AM, A, PM, or P to specify morning or afternoon. If an hour larger than 12 is entered, a 24-hour clock is assumed. 12:00 AM counts as midnight and 12:00 PM counts as noon. The time is stored as the number of seconds since midnight. The value of midnight is 0.

Examples

The following examples show the effect of some **MT** conversion codes with the **Iconv** function:

Conversion Expression
Internal Value

```
X = Iconv("02:46", "MT")
       X = 9960

X = Iconv("02:46:40am", "MTHS")
       X = 10000

X = Iconv("02:46am", "MTH")
       X = 9960

X = Iconv("02.46", "MT.")
       X = 9960

X = Iconv("02:46:40", "MTS")
       X = 10000
```

The following examples show the effect of some **MT** conversion codes with the **Oconv** function:

Conversion Expression
Internal Value

```
X = Oconv("02:46", "MT")
       X = "02:46"

X = Oconv("02:46:40am", "MTHS")
       X = "02:46:40am"

X = Oconv("02:46am", "MTH")
       X = "02:46am"

X = Oconv("02.46", "MT.")
       X = "02.46"

X = Oconv("02:46:40", "MTS")
       X = "02:46:40"
```

MUOC

Returns the internal storage value of a string as four-digit hexadecimal strings.

Syntax

```
MUOC
```

Remarks

On output, using **Oconv**, the supplied string is returned with each character converted to its four-digit hexadecimal internal storage value.

On input, using **Iconv**, the supplied string is treated as groups of four hexadecimal digits and the internal storage value is returned. Any group that comprises fewer than four digits is padded with zeros on the left.

Example

```
X = UniChar(222):UniChar(240):@FM
XInt = Oconv(X, 'MX0C')
Y = Oconv(X, 'NLSISO8859-1')
YExt = Oconv(Y, 'MX0C')
Yint = OCONV(X, 'MU0C')
```

The variables contain:

```
Xint (Internal form in hex bytes): C39EC3B0FE
Yext (External form in hex bytes): DEF03F
Yint (Internal form in UNICODE  ): 00DE00F0F8FE
```

MX

Converts hexadecimal numbers to decimal, or an ASCII value for storage, or vice versa, for output.

Syntax

```
MX [ 0C ]
```

0C converts the hexadecimal number to its equivalent ASCII character on input, and vice versa on output.

Remarks

Characters outside of the ranges 0 through 9, A through F, or a through f, cause an error.

Examples

The following examples show the effect of some **MX** conversion codes with the **Iconv** function:

Conversion Expression
Internal Value

```
X = Iconv("400", "MX")
       X = 1024

X = Iconv("434445", "MX0C")
       X = "CDE"
```

The following examples show the effect of some **MX** conversion codes with the **Oconv** function:

Conversion Expression
External Value

```
X = Oconv("1024", "MX")
       X = "400"

X = Oconv("CDE", "MX0C")
       X = "434445"
```

MY

Converts ASCII characters to hexadecimal values on input, and vice versa on output.

Syntax

```
MY
```

Remarks

Characters outside of the ranges 0 through 9, A through F, or a through f, cause an error.

Examples

The following examples show the effect of some **MY** conversion codes with the **Iconv** function:

Conversion Expression
Internal Value

```
X = Iconv("ABCD", "MY")
       X = 41424344

X = Iconv("0123", "MY")
       X = 30313233
```

The following examples show the effect of some **MY** conversion codes with the **Oconv** function:

Conversion Expression
External Value

```
X = Oconv("41424344", "MY")
       X = "ABCD"

X = Oconv("30313233", "MY")
       X = "0123"
```

NL

In NLS mode, converts numbers in a local character set to Arabic numerals.

Syntax

```
NL
```

Example

The following example shows the effect of the **NL** conversion code with the **Oconv** and **Iconv** functions.

Convert for display purposes:

```
Internal.Number = 1275
External.Number = OCONV(Internal.Number, "NL")
```

Convert for arithmetic:

```
Internal.Number = ICONV(External.Number, "NL")
```

NLS

In NLS mode, converts between the internal character set and the external character set.

Syntax

NLS *mapname*

mapname is the name of the character set map to use for the conversion.

Remarks

On output using the **Oconv** function, the **NLS** conversion code maps a string from the internal character set to the external character set specified in *mapname*.

On input using the **Iconv** function, the **NLS** conversion code assumes that the supplied string is in the character set specified by *mapname*, and maps it to the internal character set. If *mapname* is set to Unicode, the supplied string is assumed to comprise 2-byte Unicode characters. If there is an odd number of bytes in the string, the last byte is replaced with the Unicode replacement character and the value returned by the **Status** function is set to 3.

NR

Converts Arabic numerals to Roman numerals on output, and vice versa on input.

Syntax

NR

Remarks

These are the equivalent values of Roman and Arabic numerals:

Roman	Arabic
i	1
v	5
x	10
l	50
c	100
d	500
m	1000
V	5000
X	10000
L	50000
C	100000
D	500000
M	1000000

Examples

The following examples show the effect of some **NR** conversion codes with the **Iconv** function:

Conversion Expression
Internal Value

```
X = Iconv("mcmxcvii", "NR")
       X = 1997
```

```
X = Iconv("MCMXCVmm", "NR")
       X = 1997000
```

The following examples show the effect of some **NR** conversion codes with the **Oconv** function:

```
Conversion Expression
       External Value

X = Oconv(1997, "NR")
       X = "mcmxcvii"

X = Oconv(1997000, "NR")
       X = "MCMXCVmm"
```

P

Extracts data that matches a pattern.

Syntax

P (*pattern***) [; (***pattern***) ...]**

pattern specifies the pattern to match the data to and must be enclosed in parenthesis. It can be one or more of the following codes:

- *n***N** matches *n* numeric characters. If *n* is 0, any number of numeric characters match.
- *n***A** matches *n* alphabetic characters. If *n* is 0, any number of alphabetic characters match.
- *n***X** matches *n* alphanumeric characters. If *n* is 0, any number of alphanumeric characters match.

literal is a literal string that the data must match.

; separates a series of patterns.

Remarks

If the data does not match any of the patterns, an empty string is returned.

Examples

The following examples show the effect of some **P** conversion codes with the **Iconv** function:

```
Conversion Expression
       Internal Value

X = Iconv("123456789", "P(3N-3A-3X);(9N)")
       X = "123456789"

X = Iconv("123-ABC-A7G", "P(3N-3A-3X);(9N)")
       X = "123-ABC-A7G"

X = Iconv("123-45-6789", "P(3N-2N-4N)")
       X = "123-45-6789"
```

The following examples show the effect of some **P** conversion codes with the **Oconv** function:

```
Conversion Expression
       External Value

X = Oconv("123456789", "P(3N-3A-3X);(9N)")
       X = "123456789"
```

```
X = Oconv("123-ABC-A7G", "P(3N-3A-3X);(9N)")
       X = "123-ABC-A7G"

X = Oconv("ABC-123-A7G", "P(3N-3A-3X);(9N)")
       X = ""

X = Oconv("123-45-6789", "P(3N-2N-4N)")
       X = "123-45-6789"

X = Oconv("123-456-789", "P(3N-2N-4N)")
       X = ""

X = Oconv("123-45-678A",
       "P(3N-2N-4N)") X = ""
```

R

Retrieves data within a range.

Syntax

R*n*,*m* [**;** *n*,*m* ...]

n specifies the lower limit of the range.

m specifies the upper limit of the range.

; separates multiple ranges.

Remarks

If the data does not meet the range specifications, an empty string is returned.

Examples

The following example shows the effect of the **R** (Range Check) conversion code with the **Iconv** function.

```
Conversion Expression
       Internal Value

X = Iconv("123", "R100,200")
       X = 123
```

The following example shows the effect of the **R** (Range Check) conversion code with the **Oconv** function.

```
Conversion Expression
       External Value

X = Oconv(123, "R100,200")
       X = 123

X = Oconv(223, "R100,200")
       X = ""

X = Oconv(3.1E2, "R100,200;300,400")
       X = 3.1E2
```

S

Generates phonetic codes that can be used to compare words based on how they sound.

Syntax

```
S
```

Remarks

The phonetic code consists of the first letter of the word followed by a number. Words that sound similar, for example fare and fair, generate the same phonetic code.

Examples

The following examples show the effect of some **S** conversion codes with the **Iconv** function:

Conversion Expression
Internal Value

```
X = Iconv("GREEN", "S")
       X = "G650
X = Iconv("greene", "S")
       X = "G650"
X = Iconv("GREENWOOD", "S"
       X = "G653"
X = Iconv("GREENBAUM", "S")
       X = "G651"
```

TI

In NLS mode, converts times in internal format to the default locale convention format.

Syntax

```
TI
```

Example

The following example shows the effect of the **TI** conversion code with the **Oconv** function:

```
Internal.Time = TIME()International.Time = OCONV(Internal.Time,
→ "TI")
```

Chapter 8. Built-In Transforms and Routines

These topics describe the built-in transforms and routines supplied with IBM InfoSphere DataStage.

When you edit a Transformer stage, you can convert your data using one of the built-in transforms supplied with InfoSphere DataStage. Alternatively, you can convert your data using your own custom transforms. Custom transforms can convert data using functions or routines.

For more information about editing a Transformer stage, see "Transformer Stages" on page 98. For details about how to write a user-written routine or a custom transform, see Chapter 6, "Programming in IBM InfoSphere DataStage," on page 125. For a complete list of the supported BASIC functions, see Chapter 7, "BASIC Programming," on page 137.

Built-In Transforms

You can view the definitions of the built-in transforms using the Designer client.

Using IBM InfoSphere DataStage in an NLS environment has implications for some of the Data and Data Type transforms. If NLS is enabled, you should check the descriptions of the transforms in the Designer client before you use them to ensure that they will operate as required.

String Transforms

Transform	Input Type	Output Type	Folder	Description
CAPITALS	String	String	Built-in/String	Each word in the argument has its first character replaced with its uppercase equivalent if appropriate. Any sequence of characters between space characters is taken as a word, for example: `CAPITALS("monday feb 14th") => "Monday Feb 14th"`
DIGITS	String	String	Built-in/String	Returns a string from which all characters other than the digits 0 through 9 have been removed, for example: `DIGITS("123abc456") => "123456"`
LETTERS	String	String	Built-in/String	Returns a string from which all characters except letters have been removed, for example: `LETTERS("123abc456") => "abc"`

Transform	Input Type	Output Type	Folder	Description
StringDecode	see description	String	sdk/String	Loads an array for lookup purposes. The array contains *name=value* pairs. On the first call the array is saved, on all calls the supplied *name* is searched for in the array, and the corresponding *value* is returned. Takes a lookup key and an array as arguments, returns a string.
StringIsSpace	String	String	sdk/String	Returns a 1 if the string consists solely of one or more spaces.
StringLeftJust	String	String	sdk/String	Removes leading spaces from the input, and returns a string of the same length as the input. It does not reduce spaces between non-blank characters.
StringRightJust	String	String	sdk/String	Removes trailing spaces from the input, and returns a string of the same length as the input. It does not reduce spaces between non-blank characters.
StringUpperFirst	String	String	sdk/String	Returns the input string with initial caps in every word.

Date Transforms

Transform	Input Type	Output Type	Folder	Description
MONTH.FIRST	MONTH.TAG	Date	Built-in/Dates	Returns a numeric internal date corresponding to the first day of a month given in MONTH.TAG format (*YYYY-MM*), for example: `MONTH.FIRST("1993-02") => 9164` where 9164 is the internal representation of February 1, 1993.
MONTH.LAST	MONTH.TAG	Date	Built-in/Dates	Returns a numeric internal date corresponding to the last day of a month given in MONTH.TAG format (*YYYY-MM*), for example: `MONTH.LAST("1993-02") => 9191` where 9191 is the internal representation of February 28, 1993.

Transform	Input Type	Output Type	Folder	Description
QUARTER.FIRST	QUARTER.TAG	Date	Built-in/Dates	Returns a numeric internal date corresponding to the first day of a quarter given in QUARTER.TAG format (*YYYYQn*), for example: `QUARTER.FIRST ("1993Q2") => 9133` where 9133 is the internal representation of January 1, 1993.
QUARTER.LAST	QUARTER.TAG	Date	Built-in/Dates	Returns a numeric internal date corresponding to the last day of a quarter given in QUARTER.TAG format (*YYYYQn*), for example: `QUARTER.LAST("1993Q2") => 9222` where 9222 is the internal representation of March 31, 1993.
TIMESTAMP. TO.DATE	Timestamp	Date	Built-in/Dates	Converts Timestamp format (*YYYY-MM-DD HH:MM:SS*) to Internal Date format, for example: `TIMESTAMP.TO.DATE("1996-12-05 13:46:21") => "10567"`
TAG.TO.DATE	DATE.TAG	Date	Built-in/Dates	Converts a string in format *YYYY-MM-DD* to a numeric internal date, for example: `TAG.TO.DATE("1993-02-14") => 9177`
WEEK.FIRST	WEEK.TAG	Date	Built-in/Dates	Returns a numeric internal date corresponding to the first day (Monday) of a week given in WEEK.TAG format (*YYYYWnn*), for example: `WEEK.FIRST("1993W06") => 9171` where 9171 is the internal representation of February 8, 1993.

Transform	Input Type	Output Type	Folder	Description
WEEK.LAST	WEEK.TAG	Date	Built-in/Dates	Returns a numeric internal date corresponding to the last day (Sunday) of a week given in WEEK.TAG format (*YYYYWnn*), for example: `WEEK.LAST("1993W06") => 9177` where 9177 is the internal representation of February 14, 1993.
YEAR.FIRST	YEAR.TAG	Date	Built-in/Dates	Returns a numeric internal date corresponding to the first day of a year given in YEAR.TAG format (*YYYY*), for example: `YEAR.FIRST("1993") => 9133` where 9133 is the internal representation of January 1, 1993.
YEAR.LAST	YEAR.TAG	Date	Built-in/Dates	Returns a numeric internal date corresponding to the last day of a year given in YEAR.TAG format (*YYYY*), for example: `YEAR.LAST("1993") => 9497` where 9497 is the internal representation of December 31, 1993.
TIMESTAMP. TO.TIME	Timestamp	Time	Built-in/Dates	Converts TIMESTAMP format (*YYYY-MM-DD HH:MM:SS*) to internal time format. For example: `TIMESTAMP.TO.TIME("1996-12-05 13:46:21") => "49581"` where 49581 is the internal representation of December 5 1996, 1.46 p.m. and 21 seconds.
TIMESTAMP	Date	Time stamp	Built-in/Dates	Converts internal date format to TIME-STAMP format (*YYYY-MM-DD HH:MM:SS*). For example: `TIMESTAMP("10567") => "1996-12- 05 00:00:00"` where 10567 is the internal representation of December 5 1996.

Transform	Input Type	Output Type	Folder	Description
DATE.TAG	Date	DATE.TAG	Built-in/Dates	Converts a numeric internal date to a string in DATE.TAG format (*YYYY-MM-DD*), for example: `DATE.TAG(9177) => "1993-02-14"`
TAG.TO.WEEK	DATE.TAG	WEEK.TAG	Built-in/Dates	Converts a string in DATE.TAG format (*YYYY-MM-DD*) to WEEK.TAG format (*YYYYWnn*), for example: `TAG.TO.WEEK("1993-02-14") => "1993W06"`
WEEK.TAG	Date	WEEK.TAG	Built-in/Dates	Converts a date in internal date format to a WEEK.TAG string (*YYYYWnn*), for example: `WEEK.TAG(9177) => "1993W06"`
MONTH.TAG	Date	MONTH.TAG	Built-in/Dates	Converts a numeric internal date to a string in MONTH.TAG format (*YYYY-MM*), for example: `MONTH.TAG(9177) => "1993-02"`
TAG.TO.MONTH	DATE. TAG	MONTH.TAG	Built-in/Dates	Converts a string in DATE.TAG format (*YYYY-MM-DD*) to MONTH.TAG format (*YYYY-MM*), for example: `TAG.TO.MONTH("1993-02014") => "1993-02"`
QUARTER.TAG	Date	QUARTER.TAG	Built-in/Dates	Converts a numeric internal date to a string in QUARTER.TAG format (*YYYYQn*), for example: `QUARTER.TAG(9177) => "1993Q2"`
TAG.TO. QUARTER	DATE.TAG	QUARTER.TAG	Built-in/Dates	Converts a string in DATE.TAG format (*YYYY-MM-DD*) to QUARTER.TAG format (*YYYYQn*), for example: `TAG.TO.QUARTER("1993-02-14") => "1993Q2"`

Transform	Input Type	Output Type	Folder	Description
MONTH.TO. QUARTER	MONTH.TAG	QUARTER.TAG	Built-in/Dates	Converts a string in MONTH.TAG format (*YYYY-MM*) to QUARTER.TAG format (*YYYYQn*), for example: `MONTH.TO.QUARTER("1993-02") => "1993Q1"`
YEAR.TAG	Date	YEAR.TAG	Built-in/Dates	Converts a date in internal Date format to YEAR.TAG format (*YYYY*), for example: `YEAR.TAG(9177) => "1993"`
TAG.TO.YEAR	DATE.TAG	YEAR.TAG	Built-in/Dates	Converts a string in DATE.TAG format (*YYYY-MM-DD*) to YEAR.TAG format (*YYYY*), for example: `TAG.TO.YEAR("1993-02-14") => "1993"`
MONTH.TO.YEAR	MONTH.TAG	YEAR.TAG	Built-in/Dates	Converts a string in MONTH.TAG format (*YYYY-MM*) to YEAR.TAG format (*YYYY*), for example: `MONTH.TO.YEAR("1993-02") => "1993"`
QUARTER.TO. YEAR	QUARTER.TAG	YEAR.TAG	Built-in/Dates	Converts a string in QUARTER.TAG format (*YYYYQn*) to YEAR.TAG format (*YYYY*), for example: `QUARTER.TO.YEAR("1993Q2") => "1993"`
DateCurrent DateTime	-	String	sdk/date	Returns the current date/time in *YYYY-MM-DD HH:MM:SS.SSS* format.
DateCurrent GMTTime	-	String	sdk/date	Returns the current GMT date/time in *YYYY-MM-DD HH:MM:SS.SSS* format.
DateCurrent SwatchTime	-	Number	sdk/date	Returns the current Swatch or Internet time.
DateDaysSince 1900To TimeStamp	-	String	sdk/date	Converts days since 1900 into *YYYYMMDD HH:MM:SS:SSS* format.
DateDaysSince 1970To TimeStamp	-	String	sdk/date	Converts days since 1970 into *YYYYMMDD HH:MM:SS:SSS* format.

Transform	Input Type	Output Type	Folder	Description
The following transforms accept input date strings in any of the following formats: • Any delimited date giving *Date Month Year* (for example, 4/19/1999, 4.19.1999, 4/19/99, 4.19.99) • Alpha month dates (for example, Apr 08 1999, Apr 08 99) • Nondelimited dates in Year Month Date (for example, 19990419, 990419) • Julian year dates (for example, 99126, 1999126)				
DateGenericGetDay	String	String	sdk/date/ generic	Returns the Day value of the given date in *YYYYMMDD HH:MM:SS:SSS* format.
DateGeneric GetMonth	String	String	sdk/date/ generic	Returns the Month value of the given date in *YYYYMMDD HH:MM:SS:SSS* format.
DateGenericGetTime	String	String	sdk/date/ generic	Returns the Time value of the given date in *YYYYMMDD HH:MM:SS:SSS* format.
DateGeneric GetTimeHour	String	String	sdk/date/ generic	Returns the Hour value of the given date in *YYYYMMDD HH:MM:SS:SSS* format.
DateGenericGetTime Minute	String	String	sdk/date/ generic	Returns the Minute value of the given date in *YYYYMMDD HH:MM:SS:SSS* format.
DateGenericGetTime Second	String	String	sdk/date/ generic	Returns the Second value of the given date in *YYYYMMDD HH:MM:SS:SSS* format.
DateGenericGetYear	String	String	sdk/date/ generic	Returns the Year value of the given date in *YYYYMMDD HH:MM:SS:SSS* format.
DateGeneric ToInfCLI	String	String	sdk/date/ generic	Returns the input date suitable for loading using Informix® CLI.
DateGeneric ToInfCLIWithTime	String	String	sdk/date/ generic	Returns the input date formatted as suitable for loading using Informix CLI with *HH:MM:SS:SSS* at the end.
DateGeneric ToInternal	String	Date	sdk/date/ generic	Returns the input date formatted in IBM InfoSphere DataStage internal format.
DateGeneric ToInternalWithTime	String	Date	sdk/date/ generic	Returns the input date formatted in InfoSphere DataStage internal format with *HH:MM:SS.SSS* at the end.
DateGeneric ToODBC	String	String	sdk/date/ generic	Returns the input date in a format suitable for loading using ODBC stage.

Transform	Input Type	Output Type	Folder	Description
DateGeneric ToODBCWithTime	String	String	sdk/date/ generic	Returns the input date formatted for loading using ODBC stage with *HH:MM:SS.SSS* at the end.
DateGeneric ToOraOCI	String	String	sdk/date/ generic	Returns the input date formatted for loading using Oracle OCI.
DateGeneric ToOraOCIWithTime	String	String	sdk/date/ generic	Returns the input date formatted for loading using Oracle OCI with *HH:MM:SS* at the end.
DateGeneric ToSybaseOC	String	String	sdk/date/ generic	Returns the input date in a format suitable for loading using Sybase Open Client.
DateGeneric ToSybaseOC WithTime	String	String	sdk/date/ generic	Returns the input date in a format suitable for loading using Sybase Open Client with *HH:MM:SS.SSS* at the end.
DataGeneric ToTimeStamp	String	String	sdk/date/ generic	Returns the input date formatted in *YYYYMMDD HH:MM:SS:SSS* format.
DateGeneric DateDiff	String, String	String	sdk/date/ generic	Compares two dates and returns the number of days difference.
DateGeneric DaysSince1900	String	String	sdk/date/ generic	Compares the input date with 1899-12-31 midnight and returns the number of days difference.
DateGeneric DaysSince1970	String	String	sdk/date/ generic	Compares the input date with 1969-12-31 midnight and returns the number of days difference.
DateGeneric DaysSinceToday	String	String	sdk/date/ generic	Compares the input date with today midnight and returns the number of days difference.
DateGenericIsDate	String	String	sdk/date/ generic	Returns 1 if input is valid date, or 0 otherwise.
The following transforms accept delimited input date strings in the format [*YY*]*YY MM DD* using any delimiter. The strings can also contain a time entry in the format *HH:MM:SS:SSS*, *HH:MM:SS* or *HH:MM*.				
DateYearFirst GetDay	String	String	sdk/date/ YearFirst	Returns the Day value of the given date in *YYYYMMDD HH:MM:SS:SSS* format.
DateYearFirst GetMonth	String	String	sdk/date/ YearFirst	Returns the Month value of the given date in *YYYYMMDD HH:MM:SS:SSS* format.
DateYearFirst GetTime	String	String	sdk/date/ YearFirst	Returns the Time value of the given date in *YYYYMMDD HH:MM:SS:SSS* format.

Transform	Input Type	Output Type	Folder	Description
DateYearFirst GetTimeHour	String	String	sdk/date/ YearFirst	Returns the Hour value of the given date in *YYYYMMDD HH:MM:SS:SSS* format.
DateYearFirst GetTimeMinute	String	String	sdk/date/ YearFirst	Returns the Minute value of the given date in *YYYYMMDD HH:MM:SS:SSS* format.
DateYearFirst GetTimeSecond	String	String	sdk/date/ YearFirst	Returns the Second value of the given date in *YYYYMMDD HH:MM:SS:SSS* format.
DateYearFirst GetYear	String	String	sdk/date/ YearFirst	Returns the Year value of the given date in *YYYYMMDD HH:MM:SS:SSS* format.
DateYearFirst ToInfCLI	String	String	sdk/date/ YearFirst	Returns the input date in a format suitable for loading using Informix CLI.
DateYearFirst ToInfCLIWithTime	String	String	sdk/date/ YearFirst	Returns the input date formatted as suitable for loading using Informix CLI with *HH:MM:SS.SSS* at the end.
DateYearFirst ToInternal	String	Date	sdk/date/ YearFirst	Returns the input date formatted in InfoSphere DataStage internal format.
DateYearFirst ToInternalWithTime	String	Date	sdk/date/ YearFirst	Returns the input date formatted in InfoSphere DataStage internal format with *HH:MM:SS.SSS* at the end.
DateYearFirst ToODBC	String	String	sdk/date/ YearFirst	Returns the input date in a format suitable for loading using ODBC stage.
DateYearFirst ToODBCWithTime	String	String	sdk/date/ YearFirst	Returns the input date in a format suitable for loading using ODBC stage with *HH:MM:SS.SSS* at the end.
DateYearFirst ToOraOCI	String	String	sdk/date/ YearFirst	Returns the input date in a format suitable for loading using Oracle OCI.
DateYearFirst ToOraOCIWithTime	String	String	sdk/date/ YearFirst	Returns the input date in a format suitable for loading using Oracle OCI with *HH:MM:SS* at the end.
DateYearFirst ToSybaseOC	String	String	sdk/date/ YearFirst	Returns the input date in a format suitable for loading using Sybase Open Client.
DateYearFirst ToSybaseOC WithTime	String	String	sdk/date/ YearFirst	Returns the input date in a format suitable for loading using Sybase Open Client with *HH:MM:SS.SSS* at the end.

Transform	Input Type	Output Type	Folder	Description
DataYearFirst ToTimeStamp	String	String	sdk/date/ YearFirst	Returns the input date formatted in *YYYYMMDD HH:MM.SS:SSS* format.
DateYearFirstDiff	String, String	String	sdk/date/ YearFirst	Compares two dates and returns the number of days difference.
DateYearFirst DaysSince1900	String	String	sdk/date/ YearFirst	Compares the input date with 1899-12-31 midnight and returns the number of days difference.
DateYearFirst DaysSince1970	String	String	sdk/date/ YearFirst	Compares the input date with 1969-12-31 midnight and returns the number of days difference.
DateYearFirst DaysSinceToday	String	String	sdk/date/ YearFirst	Compares the input date with today midnight and returns the number of days difference.
DateYearFirstIsDate	String	String	sdk/date/ YearFirst	Returns 1 if input is valid date, or 0 otherwise.

Data Type Transforms

Transform	Input Type	Output Type	Folder	Description
DataTypeAsciiPic9	String	Number	sdk/Data Type	Converts ASCII PIC 9(n) into an integer.
DataTypeAsciiPic9V9	String	Number	sdk/Data Type	Converts ASCII PIC 9(n) with one assumed decimal place into a number with one actual decimal place.
DataTypeAscii Pic9V99	String	Number	sdk/Data Type	Converts ASCII PIC 9(n) with two assumed decimal places into a number with two actual decimal places.
DataTypeAscii Pic9V999	String	Number	sdk/Data Type	Converts ASCII PIC 9(n) with three assumed decimal places into a number with three actual decimal places.
DataTypeAscii Pic9V9999	String	Number	sdk/Data Type	Converts ASCII PIC 9(n) with four assumed decimal places into a number with four actual decimal places.
DataTypeAscii toEbcdic	String	String	sdk/Data Type	Converts ASCII string to EBCDIC.

Transform	Input Type	Output Type	Folder	Description
DataTypeEbcdicPic9	String	Number	sdk/Data Type	Converts EBCDIC PIC 9(n) into an integer.
DataTypeEbcdic Pic9V9	String	Number	sdk/Data Type	Converts EBCDIC PIC 9(n) with one assumed decimal place into a number with one actual decimal place.
DataTypeEbcdic Pic9V99	String	Number	sdk/Data Type	Converts EBCDIC PIC 9(n) with two assumed decimal places into a number with two actual decimal places.
DataTypeEbcdic Pic9V999	String	Number	sdk/Data Type	Converts EBCDIC PIC 9(n) with three assumed decimal places into a number with three actual decimal places.
DataTypeEbcdic Pic9V9999	String	Number	sdk/Data Type	Converts EBCDIC PIC 9(n) with four assumed decimal places into a number with four actual decimal places.
DataTypeEbcdic toAscii	String	String	sdk/Data Type	Converts EBCDIC string to ASCII.
DataTypePic9	String	Number	sdk/Data Type	Converts ASCII or EBCDIC PIC 9(n) into an integer.
DataTypePic9V9	String	Number	sdk/Data Type	Converts ASCII or EBCDIC PIC 9(n) with one assumed decimal place into a number with one actual decimal place.
DataTypePic9V99	String	Number	sdk/Data Type	Converts ASCII or EBCDIC PIC 9(n) with two assumed decimal places into a number with two actual decimal places.
DataTypePic9V999	String	Number	sdk/Data Type	Converts ASCII or EBCDIC PIC 9(n) with three assumed decimal places into a number with three actual decimal places.

Transform	Input Type	Output Type	Folder	Description
DataTypePic9 V9999	String	Number	sdk/Data Type	Converts ASCII or EBCDIC PIC 9(n) with four assumed decimal places into a number with four actual decimal places.
DataTypePicComp	String	Number	sdk/Data Type	Converts COBOL PIC COMP into an integer.
DataTypePicComp1	String	Number	sdk/Data Type	Converts COBOL PIC COMP-1 into a real number.
DataTypePicComp2	String	Number	sdk/Data Type	Converts COBOL PIC COMP-2 into a real number.
DataTypePicComp3	String	Number	sdk/Data Type	Converts COBOL PIC COMP-3 signed packed decimal into an integer.
DataTypePicComp3 Unsigned	String	Number	sdk/Data Type	Converts COBOL PIC COMP-3 unsigned packed decimal into an integer.
DataTypePicComp3 UnsignedFast	String	Number	sdk/Data Type	Converts COBOL PIC COMP-3 unsigned packed decimal into an integer.
DataTypePicComp3 V9	String	Number	sdk/Data Type	Converts COBOL PIC COMP-3 signed packed decimal with one assumed decimal place into a number with one actual decimal place.
DataTypePicComp3 V99	String	Number	sdk/Data Type	Converts COBOL PIC COMP-3 signed packed decimal with two assumed decimal places into a number with two actual decimal places.
DataTypePicComp3 V999	String	Number	sdk/Data Type	Converts COBOL PIC COMP-3 signed packed decimal with three assumed decimal places into a number with three actual decimal places.
DataTypePicComp3 V9999	String	Number	sdk/Data Type	Converts COBOL PIC COMP-3 signed packed decimal with four assumed decimal places into a number with four actual decimal places.

Transform	Input Type	Output Type	Folder	Description
DataTypePicComp Unsigned	String	Number	sdk/Data Type	Converts unsigned binary into an integer.
DataTypePicS9	String	Number	sdk/Data Type	Converts zoned right decimal COBOL PIC S9(n) Data type in ASCII or EBCDIC into an integer.

Key Management Transforms

Transform	Input Type	Output Type	Folder	Description
KeyMgtGetMaxKey	String, String, String, String	String	sdk/KeyMgt	Takes a column, table, ODBC stage, and a number from 1 to 99 as a unique (within the job) handle. Returns the maximum value from the specified column. Typically used for key management.
KeyMgtGetNextValue	Literal string	String	sdk/KeyMgt	Generates sequential numbers.
KeyMgtGetNextValue Concurrent	Literal string	String	sdk/KeyMgt	Generates sequential numbers in a concurrent environment.

Measurement Transforms - Area

Transform	Input Type	Output Type	Folder	Description
MeasureAreaAcresToSqFeet	String	String	sdk/Measure/ Area	Converts acres to square feet.
MeasureAreaAcresToSqMeters	String	String	sdk/Measure/ Area	Converts acres to square meters.
MeasureAreaSqFeetToAcres	String	String	sdk/Measure/ Area	Converts square feet to acres.
MeasureAreaSqFeetToSqInches	String	String	sdk/Measure/ Area	Converts square feet to square inches.
MeasureAreaSqFeetToSqMeters	String	String	sdk/Measure/ Area	Converts square feet to square meters.
MeasureAreaSqFeetToSqMiles	String	String	sdk/Measure/ Area	Converts square feet to square miles.
MeasureAreaSqFeetToSqYards	String	String	sdk/Measure/ Area	Converts square feet to square yards.
MeasureAreaSqInchesToSqFeet	String	String	sdk/Measure/ Area	Converts square inches to square feet.
MeasureAreaSqInchesToSqMeters	String	String	sdk/Measure/ Area	Converts square inches to square meters.
MeasureAreaSqMeterToAcres	String	String	sdk/Measure/ Area	Converts square metres to acres.

Transform	Input Type	Output Type	Folder	Description
MeasureAreaSqMetersToSqFeet	String	String	sdk/Measure/ Area	Converts square metres to square feet.
MeasureAreaSqMetersToSqInches	String	String	sdk/Measure/ Area	Converts square metres to square inches.
MeasureAreaSqMetersToSqMiles	String	String	sdk/Measure/ Area	Converts square metres to square miles.
MeasureAreaSqMetersToSqYards	String	String	sdk/Measure/ Area	Converts square metres to square yards.
MeasureAreaSqMilesToSqFeet	String	String	sdk/Measure/ Area	Converts square miles to square feet.
MeasureAreaSqMilesToSqMeters	String	String	sdk/Measure/ Area	Converts square miles to square meters.
MeasureAreaSqYardsToSqFeet	String	String	sdk/Measure/ Area	Converts square yards to square feet.
MeasureAreaSqYardsToSqMeters	String	String	sdk/Measure/ Area	Converts square yards to square meters.

Measurement Transforms - Distance

Transform	Input Type	Output Type	Folder	Description
MeasureDistance FeetToInches	String	String	sdk/Measure/ Distance	Converts feet to inches.
MeasureDistance FeetToMeters	String	String	sdk/Measure/ Distance	Converts feet to meters.
MeasureDistance FeetToMiles	String	String	sdk/Measure/ Distance	Converts feet to miles.
MeasureDistance FeetToYards	String	String	sdk/Measure/ Distance	Converts feet to yards.
MeasureDistance InchesToFeet	String	String	sdk/Measure/ Distance	Converts inches to feet.
MeasureDistance InchesToMeters	String	String	sdk/Measure/ Distance	Converts inches to meters.
MeasureDistance InchesToMiles	String	String	sdk/Measure/ Distance	Converts inches to miles.
MeasureDistance InchesToYards	String	String	sdk/Measure/ Distance	Converts inches to yards.
MeasureDistance MetersToFeet	String	String	sdk/Measure/ Distance	Converts meters to feet.
MeasureDistance MetersToInches	String	String	sdk/Measure/ Distance	Converts meters to inches.
MeasureDistance MetersToMile	String	String	sdk/Measure/ Distance	Converts meters to miles.
MeasureDistance MetersToYard	String	String	sdk/Measure/ Distance	Converts meters to yards.

Transform	Input Type	Output Type	Folder	Description
MeasureDistance MilesToFeet	String	String	sdk/Measure/ Distance	Converts miles to feet.
MeasureDistance MilesToInches	String	String	sdk/Measure/ Distance	Converts miles to inches.
MeasureDistance MilesToMeters	String	String	sdk/Measure/ Distance	Converts miles to meters.
MeasureDistance MilesToYards	String	String	sdk/Measure/ Distance	Converts miles to yards.
MeasureDistance YardsToFeet	String	String	sdk/Measure/ Distance	Converts yards to feet.
MeasureDistance YardsToInches	String	String	sdk/Measure/ Distance	Converts yards to inches.
MeasureDistance YardsToMeters	String	String	sdk/Measure/ Distance	Converts yards to meters.
MeasureDistance YardsToMiles	String	String	sdk/Measure/ Distance	Converts yards to miles.

Measurement Transforms - Temperature

Transform	Input Type	Output Type	Folder	Description
MeasureTemp CelsiusToFahrenheit	String	String	sdk/Measure/ Temp	Converts Celsius to Fahrenheit.
MeasureTemp FahrenheitToCelsius	String	String	sdk/Measure/ Temp	Converts Fahrenheit to Celsius.

Measurement Transforms - Time

Transform	Input Type	Output Type	Folder	Description
MeasureTime DaysToSeconds	String	String	sdk/Measure/Time	Converts days to seconds.
MeasureTime HoursToSeconds	String	String	sdk/Measure/Time	Converts hours to seconds.
MeasureTime IsLeapYear	String	String	sdk/Measure/Time	Returns 1 if the 4-digit year input is a leap year, or 0 otherwise.
MeasureTime MinutesTo Seconds	String	String	sdk/Measure/Time	Converts minutes to seconds.
MeasureTime SecondsToDays	String	String	sdk/Measure/Time	Converts seconds to days.
MeasureTime SecondsToHours	String	String	sdk/Measure/Time	Converts seconds to hours.
MeasureTimeSeconds ToMinutes	String	String	sdk/Measure/Time	Converts seconds to minutes.
MeasureTime SecondsToWeeks	String	String	sdk/Measure/Time	Converts seconds to weeks.
MeasureTime SecondsToYears	String	String	sdk/Measure/Time	Converts seconds to years.

Transform	Input Type	Output Type	Folder	Description
MeasureTime WeeksToSeconds	String	String	sdk/Measure/Time	Converts weeks to seconds.
MeasureTime YearsToSeconds	String	String	sdk/Measure/Time	Converts standard years to seconds.

Measurement Transforms - Volume

Transform	Input Type	Output Type	Folder	Description
MeasureVolume BarrelsLiquid ToCubicFeet	String	String	sdk/Measure/Volume	Converts US barrels (liquid) to cubic feet.
MeasureVolume BarrelsLiquid ToGallons	String	String	sdk/Measure/Volume	Converts US barrels (liquid) to US gallons.
MeasureVolume BarrelsLiquid ToLiters	String	String	sdk/Measure/Volume	Converts US barrels (liquid) to liters.
MeasureVolume BarrelsPetrol ToGallons	String	String	sdk/Measure/Volume	Converts US barrels (petroleum) to US gallons.
MeasureVolume BarrelsPetrol ToLiters	String	String	sdk/Measure/Volume	Converts US barrels (petroleum) to liters.
MeasureVolume BarrelsPetrol ToCubicFeet	String	String	sdk/Measure/Volume	Converts US barrels (petroleum) to cubic feet.
MeasureVolume CubicFeet ToBarrelsLiquid	String	String	sdk/Measure/Volume	Converts cubic feet to US barrels (liquid).
MeasureVolume CubicFeet ToBarrelsPetrol	String	String	sdk/Measure/Volume	Converts cubic feet to US barrels (petroleum).
MeasureVolume CubicFeet ToGallons	String	String	sdk/Measure/Volume	Converts cubic feet to US gallons.
MeasureVolume CubicFeet ToLiters	String	String	sdk/Measure/Volume	Converts cubic feet to liters.
MeasureVolume CubicFeet ToImpGallons	String	String	sdk/Measure/Volume	Converts cubic feet to imperial gallons.
MeasureVolume GallonsTo BarrelsLiquid	String	String	sdk/Measure/Volume	Converts US gallons to US barrels (liquid).
MeasureVolume GallonsTo BarrelsPetrol	String	String	sdk/Measure/Volume	Converts US gallons to US barrels (petroleum).
MeasureVolume GallonsTo CubicFeet	String	String	sdk/Measure/Volume	Converts US gallons to cubic feet.

Transform	Input Type	Output Type	Folder	Description
MeasureVolume GallonsToLiters	String	String	sdk/Measure/Volume	Converts US gallons to liters.
MeasureVolume LitersTo BarrelsLiquid	String	String	sdk/Measure/Volume	Converts liters to US barrels (liquid).
MeasureVolume LitersTo BarrelsPetrol	String	String	sdk/Measure/Volume	Converts liters to US barrels (petroleum).
MeasureVolume LitersTo CubicFeet	String	String	sdk/Measure/Volume	Converts liters to cubic feet.
MeasureVolume LitersToGallons	String	String	sdk/Measure/Volume	Converts liters to US gallons.
MeasureVolume LitersToGallons	String	String	sdk/Measure/Volume	Converts liters to imperial gallons.
MeasureVolume ImpGallons ToCubicFeet	String	String	sdk/Measure/Volume	Converts imperial gallons to cubic feet.
MeasureVolume ImpGallons ToLiters	String	String	sdk/Measure/Volume	Converts imperial gallons to liters.

Measurement Transforms - Weight

Transform	Input Type	Output Type	Folder	Description
MeasureWeightGrains ToGrams	String	String	sdk/Measure/ Weight	Converts grains to grams.
MeasureWeightGrams ToGrains	String	String	sdk/Measure/ Weight	Converts grams to grains.
MeasureWeightGrams ToOunces	String	String	sdk/Measure/ Weight	Converts grams to ounces.
MeasureWeightGrams ToPennyWeight	String	String	sdk/Measure/ Weight	Converts grams to penny weights.
MeasureWeightGrams ToPounds	String	String	sdk/Measure/ Weight	Converts grams to pounds.
MeasureWeightKilograms ToLongTons	String	String	sdk/Measure/ Weight	Converts kilograms to long tons.
MeasureWeightKilograms ToShortTons	String	String	sdk/Measure/ Weight	Converts kilograms to short tons.
MeasureWeightLongTons ToKilograms	String	String	sdk/Measure/ Weight	Converts long tons to kilograms.
MeasureWeightLongTons ToPounds	String	String	sdk/Measure/ Weight	Converts long tons to pounds.
MeasureWeightOunces ToGrams	String	String	sdk/Measure/ Weight	Converts ounces to grams.
MeasureWeightPennyWeight ToGrams	String	String	sdk/Measure/ Weight	Converts penny weights to grams.
MeasureWeightPounds ToGrams	String	String	sdk/Measure/ Weight	Converts pounds to grams.

Transform	Input Type	Output Type	Folder	Description
MeasureWeightPounds ToLongTons	String	String	sdk/Measure/ Weight	Converts pounds to long tons.
MeasureWeightPounds ToShortTons	String	String	sdk/Measure/ Weight	Converts pounds to short tons.
MeasureWeightShortTons ToKilograms	String	String	sdk/Measure/ Weight	Converts short tons to kilograms.
MeasureWeightShortTons ToPounds	String	String	sdk/Measure/ Weight	Converts short tons to pounds.

Numeric Transforms

Transform	Input Type	Output Type	Folder	Description
NumericIsSigned	String	String	sdk/Numeric	Returns 0 if the input is nonnumeric or zero, 1 for positive numbers, and -1 for negative numbers.
NumericRound0	String	String	sdk/Numeric	Returns the nearest whole number to the input number.
NumericRound1	String	String	sdk/Numeric	Returns the input number to the nearest 1 decimal place.
NumericRound2	String	String	sdk/Numeric	Returns the input number to the nearest 2 decimal places.
NumericRound3	String	String	sdk/Numeric	Returns the input number to the nearest 3 decimal places.
NumericRound4	String	String	sdk/Numeric	Returns the input number to the nearest 4 decimal places.

Row Processor Transforms

Transform	Input Type	Output Type	Folder	Description
RowProcCompareWith PreviousValue	String	String	sdk/RowProc	Compares the current value with the previous value. Returns 1 if they are equal, or 0 otherwise. Can only be used in one place in a job.

Transform	Input Type	Output Type	Folder	Description
RowProcGetPreviousValue	String	String	sdk/RowProc	Returns the previous value passed into this transform, preserves the current input for the next reference. Can only be used in one place in a job.
RowProcRunningTotal	String	String	sdk/RowProc	Returns the running sum of the input. Can only be used in one place in a job.

Utility Transforms

Transform	Input Type	Output Type	Folder	Description
UtilityAbortToLog	String	-	sdk/Utility	Causes the job to terminate and writes the supplied message to the critical error log in the Director client. This is intended for development use only.
UtilityRunJob	String, delimited string, number, number	Array	sdk/Utility	Runs the specified job and returns statistics from the job run. The job is specified by job name, list of parameters delimited by | characters, a row limit, and a warning limit. The statistics are returned in an array.
UtilityGetRunJobInfo	Output from Utility RunJob, String, String	String	sdk/Utility	Extracts information from UtilityRunJob output. Takes the output from UtilityRunJob, an action, and (optionally) a link name as arguments. Possible actions are: LinkCount JobName JobCompletionStatus StartTime EndTime
UtilityMessageToLog	String	-	sdk/Utility	Writes the user-supplied message to the log in the Director client.
UtilityPrintColumn ValueToLog	String	-	sdk/Utility	Writes a column value to the log in the Director client.

Transform	Input Type	Output Type	Folder	Description
UtilityPrintHexValueToLog	String	-	sdk/Utility	Converts the supplied value and processes it as a string. Converts each character in the string to its ASCII hexadecimal equivalent and writes it to the log in the Director client.
UtilityWarningToLog	String	-	-	Writes the supplied message as a warning to the log in the Director client.
UtilityHashLookup	String, String, String	String	sdk/Utility	Executes a lookup against a hash table. Takes the hash table name, hash key value, and column position as arguments. Returns the record.

Built-In Routines

There are three types of routines supplied with IBM InfoSphere DataStage:

- **Built-in before/after subroutines**. These routines are stored under the **Routines → Built-In → Before/After** folder in the repository tree. They are compiled and ready for use as a before-stage or after-stage subroutine or as a before-job or after-job routine.
- **Examples of transform functions**. These routines are stored under the **Routines → Examples → Functions** folder in the repository tree and are used by the built-in transforms supplied with InfoSphere DataStage. You can copy these routines and use them as a basis for your own user-written transform functions.
- **Transform functions used by the SDK transforms**. These are the routines used by the SDK transforms of the same name. They are stored under **Routine → sdk**. These routines are not offered by the Expression Editor and you should use the transform in preference to the routine (as described in "Built-In Transforms" on page 291).

You can view the definitions of these routines using the Designer client, but you cannot edit them. You can copy and rename them, if required, and edit the copies for your own purposes.

Built-In Before/After Subroutines

There are a number of built-in before/after subroutines supplied with IBM InfoSphere DataStage:

- **DSSendMail**. This routine is an interlude to the local send mail program.
- **DSWaitForFile**. This routine is called to suspend a job until a named job either exists, or does not exist.
- **DSJobReport**. This routine can be called at the end of a job to write a job report to a file. The routine takes an argument comprising two or three elements separated by semicolons as follows:
  - **Report type**. 0, 1, or 2 to specify report detail. Type 0 produces a text string containing start/end time, time elapsed and status of job. Type 1 is as a basic report but also contains information about individual stages and links within the job. Type 2 produces a text string containing a full XML report.
  - **Directory**. Specifies the directory in which the report will be written.

  - **XSL stylesheet**. Optionally specifies an XSL style sheet to format an XML report.

    If the job had an alias ID then the report is written to *JobName_alias*.txt or *JobName_alias*.xml, depending on report type. If the job does not have an alias, the report is written to *JobName_YYYYMMDD_HHMMSS*.txt or *JobName_YYYYMMDD_HHMMSS*.xml, depending on report type.
- **ExecDOS**. This routine executes a command via an MS-DOS shell. The command executed is specified in the routine's input argument.
- **ExecDOSSilent**. As ExecDOS, but does not write the command line to the job log.
- **ExecTCL**. This routine executes a command via an InfoSphere Information Server engine shell. The command executed is specified in the routine's input argument.
- **ExecSH**. This routine executes a command via a UNIX Korn shell.
- **ExecSHSilent**. As **ExecSH**, but does not write the command line to the job log.

These routines appear in the list of available built-in routines when you edit the **Before-stage subroutine** or **After-stage subroutine** fields in an Aggregator, Transformer, or supplemental stage, or the **Before-job subroutine** or **After-job subroutine** fields in the Job Properties dialog box.

You can also copy these routines and use the code as a basis for your own before/after subroutines.

If NLS is enabled, you should be aware of any mapping requirements when using ExecDOS and ExecSH (or ExecDOSSilent and ExecSHSilent) routines. If these routines use data in particular character sets, then it is your responsibility to map the data to or from Unicode.

Example Transform Functions

These are the example transform functions supplied with IBM InfoSphere DataStage:

- **ConvertMonth**. Transforms a MONTH.TAG input. The result depends on the value for the second argument:
  - F (the first day of the month) produces a DATE.TAG.
  - L (the last day of the month) produces a DATE.TAG.
  - Q (the quarter containing the month) produces a QUARTER.TAG.
  - Y (the year containing the month) produces a YEAR.TAG.
- **ConvertQuarter**. Transforms a QUARTER.TAG input. The result depends on the value for the second argument:
  - F (the first day of the month) produces a DATE.TAG.
  - L (the last day of the month) produces a DATE.TAG.
  - Y (the year containing the month) produces a YEAR.TAG.
- **ConvertTag**. Transforms a DATE.TAG input. The result depends on the value for the second argument:
  - I (internal day number) produces a Date.
  - W (the week containing the date) produces a WEEK.TAG.
  - M (the month containing the date) produces a MONTH.TAG.
  - Q (the quarter containing the date) produces a QUARTER.TAG.
  - Y (the year containing the date) produces a YEAR.TAG.
- **ConvertWeek**. Transforms a WEEK.TAG input to an internal date corresponding to a specific day of the week. The result depends on the value of the second argument:
  - 0 produces a Monday.
  - 1 produces a Tuesday.
  - 2 produces a Wednesday.
  - 3 produces a Thursday.

  - 4 produces a Friday.
  - 5 produces a Saturday.
  - 6 produces a Sunday.

    If the input does not appear to be a valid WEEK.TAG, an error is logged and 0 is returned.
- **ConvertYear**. Transforms a YEAR.TAG input. The result depends on the value of the second argument:
  - F (the first day of the year) produces a DATE.TAG.
  - L (the last day of year) produces a DATE.TAG.
- **QuarterTag**. Transforms a Date input into a QUARTER.TAG string (*YYYYQn*).
- **Timestamp**. Transforms a timestamp (a string in the format *YYYY-MM-DD HH*:*MM*:*SS*) or Date input. The result depends on the value for the second argument:
  - TIMESTAMP produces a timestamp with time equal to 00:00:00 from a date.
  - DATE produces an internal date from a timestamp (time part ignored).
  - TIME produces an internal time from a timestamp (date part ignored).
- **WeekTag**. Transforms a Date input into a WEEK.TAG string (*YYYY*W*nn*).

Chapter 9. Hashed File Stage Disk Caching

Prior to Release 5.1, the Hashed File stage had only one method of caching rows for both reading (reference links) and writing (output links). This method is called link private caching (formerly called stage caching). This caching mechanism is a private per-link cache. As a result, each link in a job using a Hashed File stage must allocate and manage resources to support the cache. Sharing is allowed within a job with a single data stream but not between jobs or multiple data streams in one job. This results in significant resource usage and inefficient startup times (in the case of reference links).

Also because the write cache buffers output rows until a threshold is met, cached rows are not reflected in the database until the cache is flushed. This creates a problem for job designs that use the same hashed file for both reference lookups and updates.

IBM InfoSphere DataStage provides the option to

- Cache blocks in the server's memory
- Allow the same hashed file to be referenced by multiple links
- Make inserts and updates visible to all processes that have the file open

Centralized shared-memory system disk caching, hereafter called system caching, reduces the use of system resources by implementing only one cache that can be fully configured and that supports both reading and writing.

Release 6.0 introduced an additional option called link public caching. This option allows multiple data streams within one job to use the same cache file. Link public caching was developed to take full advantage of InfoSphere DataStage's parallel engine by maximizing efficiency with a symmetric multiprocessor (SMP) when using a lookup file.

Functions that support disk caching are described in *InfoSphere DataStage Programmer's Guide*.

These topics describe user commands as well as the capabilities given to the InfoSphere DataStage administrator to adjust a number of system configuration values to maximize performance based on hardware configuration and InfoSphere DataStage steps.

Functionality

Disk caching has the following functionality and benefits:

- Supports shareable update or write file access in a single data stream in a single job (link private caching)
- Supports shareable update or write file access with
  - multiple data streams within a single job
  - multiple jobs
  - a job running with the parallel engine under SMP

  while maintaining files cached in memory (link public caching)
- Supports shareable update or write file access across jobs on one system while maintaining files cached in memory (system caching)
- Allows the exploitation of the capabilities of SMP allowing multiple concurrent data streams (link public caching)
- Supports quick in-memory access to data by an application including just updated or newly created data

- Supports in-memory access to just updated or newly created data by other processes
- Supports system tunables that allow an administrator to configure the disk cache algorithms to best meet the system configuration and expected size of files

The following functionality is not supported:

- Caching of files larger than half a terabyte
- System caching of file types 1, 19, 25 (`B tree'), or 27 (partitioned)
- System caching of existing files with separation values (block sizes) other than 1, 2, 4, 8, 16, 32 or 64
- Automatic designation of files as system cached
- Use of utilities (backup, restore, resize, and filefix) against files designated to be used by the system cache

Terminology

The following terminology is used in this document:

Term **Meaning**

block A group of records or rows. The server engine puts records that hash to the same group number into one block. Block size is determined by the CREATE.FILE's SEPARATION value.

blockset buffer
A unit of memory within the disk shared segments with a size of *n* k plus the size of the blockset head structure. *n* can be 4, 8, 16, or 32.

blockset freechain
The chain of unused blockset buffers currently available within any of the configured disk-shared segments.

cache A subsystem in which frequently used data is made accessible for quick access.

cache daemon
An asynchronous background process that does the write-defer-state writes.

cache file chain
A set of cache file entries either used (an open file) or unused.

cache file entry
A structure defining one server engine file and the related information about its state.

device number
A unique number associated with the partition (a device) on which the inode resides. See also inode number.

disk shared memory segments
The segments into which the IBM InfoSphere DataStage system cache memory is allocated. This area is then broken into blockset entries.

file A server engine native file created with the CREATE.FILE command.

flush The time when a currently allocated blockset is released and might be taken from one file and used for another set of blocks from the same or a different file.

inconsistent state
The state of a file in which some, but not all, writes generated by an application have been physically written to disk before the application terminates without a proper close file.

inode number
A unique number associated with each filename. This number is used to look up an entry in the inode table which gives information on the type, size, and location of the file and the user id of the owner of the file. See also device number.

overflow block
A unique block or set of blocks in which the overflow portion of a record's fields are stored if all of that record's data fields cannot fit in its group.

pid A unique identifier of a process.

preread
The act of reading one or more blocks of a file into cache before a request for that block.

public HEAPCHUNK
A consecutive set of blocksets (bset) allocated as one unit (128 K) to a hash file server for a piece of a link public cache.

semaphore
An operating system structure that allows processes to gate each other to single-thread through a procedure.

symmetric multiprocessing (SMP)
The processing of programs by multiple processors that share a common operating system and memory. A single copy of the operating system is in charge of all the processors. In SMP, hardware resources are typically shared among processors

write defer
A block currently in a blockset that has been modified from the image on disk and made visible to other applications, but that has not been updated on the disk file. The file is in an inconsistent state until all write-deferred blocks are written.

Multiple Data Streams

In IBM InfoSphere DataStage, multiple data streams occur in one of three states:

- Processing multiple data streams within the same job
- Processing a single large data source in a number of partitioned sets using InfoSphere DataStage's parallel engine
- Running multiple jobs that reference the same file

To gain processing efficiencies, you can process multiple data streams with a single, common, cached lookup file.

Guidelines for Choosing a Type of Caching

Use the following as a guideline as you select the type of caching to use:

To Use

Share between reference and output files in a single data stream
Link private caching

Share among multiple data streams or within a container running with the parallel engine
Link public caching

Share among multiple jobs running sequentially or in parallel using the same reference file or output file System caching

Preparing for Link Private Caching

In the Administrator client, select a project on the Projects page. Click **Properties**. On the **Tunables** tab, set the **Read cache size** (for reference files) or the **Write cache size** (for output files) to the upper limit appropriate for your job and resources. IBM InfoSphere DataStage does not use all the memory specified at once. Rather it takes memory in segments up to the limit specified. The default for each is 128 MB.

Preparing for Link Public Caching or System Caching on UNIX Platforms

By default, IBM InfoSphere DataStage is shipped with link public caching and system caching disabled. To enable disk caching, the InfoSphere DataStage administrator must perform the following steps.

1. Log in as `dsadm`.
2. Use the following command to change your current directory to the DSEngine install directory.
   ```
   cd 'cat /.dshome'
   ```
   Edit the *uvconfig* file located in the server engine directory (specified in the */.dshome* file), and set the disk cache tunables to the desired values. At the very least, set the DISKCACHE tunable to a desired size in megabytes. (See "Tuning Link Public Caching and System Caching" on page 327.)

   Note: The default values serve as a reasonable set of initial values.
3. Ensure there are no active InfoSphere DataStage client connections or interactive users.
4. Stop the server engine as follows:
   ```
   ./bin/uv -admin -stop
   ```
   Note: You cannot continue with step 5 until all InfoSphere DataStage applications have stopped running. Use the following command to verify all InfoSphere DataStage applications have stopped running:
   ```
   ./bin/uv -admin -info
   ```
   If all applications have stopped, *the output is*:
   ```
   DSEngine, rev xxxx not currently running
   ```
5. Generate a new engine configuration as follows:
   ```
   ./bin/uv -admin -regen
   ```
6. Restart the server engine as follows:
   ```
   ./bin/uv -admin -start
   ```

Link public caching and system caching is now enabled. Once caching is enabled, new or existing job designs can use this functionality. See "Using Link Public Caching" on page 318 or "Using System Caching" on page 319.

If you receive a host operating system error indicating that InfoSphere DataStage segments cannot be assigned, review information about operating system kernel parameters and make any necessary changes to them.

Special Requirements for AIX to Size the Disk Cache

Because of the default address-space model for 32-bit processes on AIX® systems, additional preparation might be needed for all of the disk caching options. The default allocation of space is 128 megabytes. The optimal maximum allocation is 512 megabytes.

If you want to allocate more than 128 megabytes of space for the disk cache on an AIX system, do the following:

1. Log in as `dsadm`.
2. Use the following command to change your current directory to the DSEngine install directory:
   ```
   cd 'cat /.dshome'
   ```
3. Edit the *uvconfig* file using a text editor such as vi.
   a. Change DMEMOFF to 0x90000000
   b. Change PMEMOFF to 0xa0000000

Save the *uvconfig* file.

4. Ensure there are no active IBM InfoSphere DataStage client connections or interactive users.
5. Stop the server engine as follows:

   ```
   . ./dsenv
   ./bin/uv -admin -stop
   ```

   Note: You cannot continue with step 6 until all InfoSphere DataStage applications have stopped running. Use the following command to verify all InfoSphere DataStage applications have stopped running:

   ```
   ./bin/uv -admin -info
   ```

   If all applications have stopped, the output is:

   ```
   DSENGINE, rev xxxx not currently running
   ```
6. Generate a new engine configuration as follows:

   ```
   ./bin/uv -admin -regen
   ```

   If the command is successful, the output is:

   ```
   uvregen: reconfiguration complete, disk segment size is xxxxxxx
   ```
7. Add the following environmental settings to the *.dsenv* file:

   ```
   LDR_CNTRL=MAXDATA=0x30000000;export LDR_CNTRL
   ```
8. Apply the new environmental settings by executing

   ```
   . ./dsenv
   ```
9. Restart the server engine as follows:

   ```
   ./bin/uv -admin -start
   ```

   Note: These settings can affect the amount of memory used for memory-intensive supplemental stages, and such stages can limit the amount of memory available for caching.

Preparing for Link Public Caching or System Caching on Windows Platforms

By default, IBM InfoSphere DataStage is shipped with link public caching and system caching disabled. To enable disk caching, the InfoSphere DataStage administrator must perform the following steps.

1. Log in as a Windows Administrator.
2. Using a text editor such as Notepad, edit the *uvconfig* file located in the server engine directory, and set the disk cache tunables to the desired values. At the very least, set the DISKCACHE tunable to a desired size in megabytes. (See "Tuning Link Public Caching and System Caching" on page 327.)

   Note: The default values serve as a reasonable set of initial values.
3. Ensure there are no active InfoSphere DataStage client connections or interactive users.
4. Stop the server engine as follows:
   a. Choose **Start → Settings → Control Panel → IBM InfoSphere Information Server**. The **InfoSphere DataStage** Control Panel dialog box appears.
   b. Click **Stop All Services** and click **Yes** in response to the message that all of the InfoSphere DataStage Services will be stopped.
   c. Click **OK** to exit the Control Panel.

   Note: You cannot continue with step 5 until all InfoSphere DataStage applications have stopped running. To verify no InfoSphere DataStage applications are running, view the **Processes** tab in the Task Manager. You should *not* find an entry called *uvsh* or any entries beginning with the letters *ds*.

Note: Generate a new engine configuration file as follows:

From a Windows NT command prompt, change to the server engine directory and issue the following command:

```
C:\IBM\InformationServer\Server\DSEngine\bin\uvregen.exe
```

where **C:\IBM\InformationServer\Server\DSEngine** is the installed server engine location.

5. Restart the server engine as follows:
   a. Choose **Start → Settings → Control Panel → IBM InfoSphere Information Server**. The **InfoSphere DataStage Control Panel** dialog box appears.
   b. Click **Start All Services** and click **Yes** in response to the message that this will start all InfoSphere DataStage services.
   c. Click **OK** to exit the InfoSphere DataStage Control Panel.

   Note: If you receive a host operating system error indicating that InfoSphere DataStage segments cannot be assigned, review information about operating system kernel parameters and make any necessary changes to them.

Link public caching and system caching are now enabled. Once caching is enabled, new or existing job designs can use this functionality. See "Using Link Public Caching" or "Using System Caching" on page 319.

Using Link Private Caching

The server engine uses private space if the following are true:

- Disk caching is enabled (either **Enabled** or **Enabled Lock for Updates** from the **Preload file to memory** drop-down list on the **Output** tab of the Hashed File Stage dialog box)
- **Enable hashed file cache sharing** is *not* selected from the **General** tab for the job (**Edit → Job Properties**) prior to the compile. As the default, **Enable hashed file cache sharing** is *not* selected.

With all of these conditions met, a new applications uses link private caching. Runtime log messages refer to link private.

If an existing application is recompiled, it might run with a different log file. A job with a single stream works the same as it did with the prior release, but runtime log messages now refer to link private.

Using Link Public Caching

The server engine uses public space if all of the following are true:

- Disk caching is enabled (either **Enabled** or **Enabled Lock for Updates** from the **Preload file to memory** drop-down list on the **Output** tab of the Hashed File Stage dialog box).
- **Enable hashed file cache sharing** is selected from the **General** tab for the job (**Edit → Job Properties**) prior to the compile. As the default, **Enable hashed file cache sharing** is *not* selected.
- Disk caching is turned on in the *uvconfig* file on the server (see "Tuning Link Public Caching and System Caching" on page 327).
- The lookup file will run in more than one stream, either in multiple data streams within the same job or in partitioned sets using the IBM InfoSphere DataStage parallel engine.

If any of the last three above is not true, link private is used for the cache.

With all of these conditions met, a new applications uses link public caching. Runtime log messages refer to link public.

To obtain the status of disk cache, see “Obtaining Status” on page 320.

Using System Caching

As an IBM InfoSphere DataStage process is initiated, the set of shared memory segments that hold the disk cache is made visible to the process. For disk caching, it is better for an administrator to use less than the maximum allowable shared disk cache memory to allow applications to run in the remaining working space.

The layout of the shared disk cache segments allows efficient, serialized update access to the list of blocks cached, on a per file (inode and device) basis.

Creating a Hash File for System Caching

To utilize system caching within the Hashed File stage, you must create a file with the caching attributes write immediate or write deferred. You can create such a file by selecting **Allow stage write cache** on the Input page of the Hashed File Stage dialog box. Specify the desired attributes through the Create file options dialog boxes. This dialog box includes the **Caching attributes** drop-down list box. The drop-down list box has the following entries:

- **NONE.** Does not assign any special caching attributes. This is the default.
- **WRITE DEFERRED.** Enables caching of the specified file using demand or lazy updating. (In the *uvconfig* file, if the DCWRITEDAEMON is enabled, then lazy updating is enabled. See “Tuning Link Public Caching and System Caching” on page 327.)
- **WRITE IMMEDIATE.** Enables caching of the specified file using synchronous writes.

Server engine commands

The IBM InfoSphere DataStage administrator can administer and monitor the system cache subsystem through the server engine command line interface as described below.

To use any of these commands, prior to running a job or subsequent to running a job, create before-job or after-job subroutine definitions. See Chapter 6, “Programming in IBM InfoSphere DataStage,” on page 125 for additional information.

Creating New and Altering Existing Hashed Files

The following command creates a new cached hashed file:

```
CREATE.FILE
```

The CREATE.FILE command has been extended by two options, WRITE.CACHE and WRITE.CACHE.DEFER, if the file is to use the shared memory disk cache. These options have the following meanings:

Option Description

WRITE.CACHE
Caches files for reads and writes with immediate writes.

WRITE.CACHE.DEFER
Caches files for reads and writes with the writes deferred until the close.

Note: The block size of the file must be 1KB, 2KB, 4KB, 8KB, 16KB, or 32KB. You control this by setting the file separation to 2, 4, 8, 16, 32, or 64, respectively.

The following command changes the mode of an existing hashed file:

```
SET.MODE filename [READ.ONLY | READ.WRITE | WRITE.CACHE |
WRITE.CACHE.DEFER | INFORM]
```

The command has the following options:

Option Description

READ.ONLY
Forces the file to be read-only, cache reads.

READ.WRITE
Restores file to normal read/write mode. This is the default value.

WRITE.CACHE
Caches files for reads and writes with immediate writes.

WRITE.CACHE.DEFER
Caches files for reads and writes with the writes deferred until the close.

INFORM
Displays the current setting of the "readonly" field in the header.

Note: The block size of the file must be 1KB, 2KB, 4KB, 8KB, 16KB, or 32KB. You control this by setting the file separation to 2, 4, 8, 16, 32, or 64, respectively.

Obtaining Status

The administrator (or user) can obtain the current status of the disk cache by using the following command:

```
LIST.FILE.CACHE [DEVICE xxx INODE yyy | FILE name | [EVERY]]
[[DETAIL][MRURO][MRUWD]]
```

Note: This command is available to both link public caching and system caching.

The command has the following options:

Option Description

DEVICE *xxx* **INODE** *yyy*
Supplies information for the cache file associated with the unique device number and inode number on which the cached file is located. *xxx* and *yyy* are decimal unless they start with `0[X|x]`, which indicates hexadecimal.

FILE *name*
Supplies information for the named cached file.

EVERY
Supplies information about all open files.

DETAIL
Supplies additional information.

MRURO
Lists all blocksets on the file cache read-only blockset queue. These blocksets are displayed at the end of all cache file entries, one entry per line.

MRUWD
Lists each blockset on the file's write-deferred queue.

If the disk cache daemon is running, the following line is displayed first.

```
DAEMON.FILE.CACHE daemon active with pause of x, pid of y
```

where *x* is the pause interval in milliseconds and y is the pid identification. For additional information, see "Starting and Stopping the Cache Daemon" on page 326.

A process can own one or more of the following semaphores: daemon request, blockset freechain, and cache file chain. If one or more is owned, the appropriate lines are output.

```
daemon request semaphore held by y
blockset freechain semaphore held by y
cache file chain semaphore held by y
```

where *y* is the pid identification.

When this command is executed, two lines show general cache status. For example,

fileentries	blocksets	freechain	flushedro	flushedwd	blockkhits
11	61	61	50	32	23292

The meaning of the status information is as follows:

Category
Meaning

fileentries
Number of cache file entries.

blocksets
Total number of blocksets in disk cache.

freechain
Number of blocksets currently available for use.

flushedro
Total number of read-only blocksets flushed.

flushedwd
Total number of write-deferred blocksets flushed.

blockhits
Total number of blocks found in cache.

If DETAIL is specified, four additional lines provide detailed disk cache status:

blocksize	arraysize	flushpc	maxpc	catpc
16384	256	80	40	50

nophyread	nophywrite	nobsetwhits
29	445	4473

The meaning of the detailed status is as follows:

Category
Meaning

blocksize
Configured size of blockset buffer. See DCBLOCKSIZE.

arraysize
Configured number of arrays per cache file entry. See DCMODULUS.

flushpc
Configured flushpc percent. See DCFLUSHPCT.

maxpc Configured maxpc percent. See DCMAXPCT.

catpc Configured catpc percent. See DCCATALOGPCT.

nophyread
Total number of reads done to the operating system.

nophywrite
Total number of writes done to the operating system.

nobsetwhits
Total number of blocksets found in cache.

The following information is provided for each file:

Device...	Inode...	open	openwd	c	t	r	d	time	fullname
8912917	1297708	1	1	C	D	W	1	11:46:17	tress/REL7.DY.1/DATA.30

	blocksets	bsetswd	flushedro	flushedwd	bsethits	hblockf
	12	3	0	144	5667	0x30000

The meaning of this information is as follows:

Category
Meaning

Device
A number that identifies the logical partition of the disk where the file system is located.

Inode A number that identifies the file that is being accessed.

open The number of current opens to this file.

openwd
The number of current write-deferred opens to this file.

c The file's catalogue status:
- C if the file is catalogued
- A space if it is not.

t The file type:
- D represents type 30 data
- O represents type 30 overflow
- S represents a IBM InfoSphere DataStage link public file
- A space represents a hashed file

r The read/write status of the file:
- R represents read only
- W represents write deferred
- A space if any other status

d The status of the cache daemon. The value is
- 1 if the cache daemon is actively monitoring the status of the file
- ? if the daemon abnormally terminated

- A space if the cache daemon is not actively monitoring the status of the file and the daemon has not abnormally terminated

time The time the file was opened.

fullname
The last 23 bytes of the full path.

blocksets
The number of blocksets currently used by this file.

bsetswd
The number of blocksets currently with at least one write-deferred block.

flushedro
The number of read-only blocksets that have flushed.

flushedwd
The number of blocksets flushed that had deferred writes.

bsethits
The number of blocksets found in the cache for this file.

hblockf
The highest block number in the file expressed in hexadecimal.

If a file-entry semaphore used for this cache file entry is currently held, a line is output with this information:

`this cache file entry semaphore (`*x*`) held by` *y*

where *x* is the number of the semaphore and *y* is the pid.

If DETAIL is specified, current file status information is displayed.

0xbaseblock	inset	mru	latch	cntovf	writedef	time
0	8	WD	0x0	0	0x80000000	11:46:17
10000	8	RO	0x0	0	0x0	11:46:17
20000	8		0x80000000	0	0x0	11:46:17

For each blockset entry, the meaning of this information is as follows:

Category
Meaning

0xbaseblock
The block number in hexadecimal (0x prefix).

inset The number of blocks in this set.

mru The read-only or write-deferred status:
- WD for the cache file's write-deferred list
- RO for the cache file's read-only list

latched
Four hexadecimal characters showing the latch settings (0x prefix), 1 bit for each block latched in the current block set from left to right.

cntovf The number of processes referencing an overflow group in this blockset.

writedef
Four hexadecimal characters showing the deferred setting (0x prefix), 1 bit for each block in the current block set from left to right.

time The time the blockset was last referenced.

If a cache file array semaphore corresponding to a displayed blockset is currently held by a process, a line is output with the following information.

```
Array entry z has cache file array semaphore (x) held by y
```

where *x* is the number of the semaphore, *y* is the pid, and *z* is the array entry number.

If DETAIL is specified and link public caching is in effect, this additional information is provided. For example:

```
bset 0x4001 first of 8 bsets make up this public HEAPCHUNK
   next 247 public HEAPCHUNKs of 8 bsets are consecutive
bset 0x7C4001 first of 2 bsets make up this public HEAPCHUNK
```

The above indicates the link public file is using 248 HEAPCHUNKs of 128 K each and 1 HEAPCHUNK of 32 K

A second form of this command is also available.

```
LIST.FILE.CACHE [DEVICE xxx INODE yyy|FILE name] BLOCK zzz [OVER.30]
```

When this command is executed, a dump of the block is displayed in hexadecimal, and record keys are listed. If an overflow block of a type 30 file is desired, enter the OVER.30 option. After two header lines, each 64 bytes of data are displayed on a line with all zero lines skipped. For each key, one line is output. The key is a value of up to 511 bytes. The dump terminates when a key is longer than 511 bytes. *xxx*, *yyy, and zzz* are decimal unless they start with 0[X|x], which indicates hexadecimal.

Changing the Status

The IBM InfoSphere DataStage administrator can change the status of the shared memory disk cache.

Note: This command is available to both link public caching and system caching.

Note: You must be logged into the dshome account to change the status. See “Logging into the dshome Account” on page 326 for information about logging into the dshome account.

The command is

```
CLEAR.FILE.CACHE
   [[FILE filename [GROUP zzz]]
   [FORCEWRITE | FLUSHRO | CLOSE | CLEAR.STAT]
| ALL| DCFILE | ENTRY | ARRAY | FREECHAIN | DAEMON]
[USER x | SEMNO y]
[STOP {DAEMON}]
[ABORT] {DETAIL}
```

The command has the following options:

Options
Description

FILE *filename*
Names the file for which the status is to be changed. If not specified, the status of all cache files is changed.

GROUP *zzz*
Identifies the group number for which the status is to be changed. If not specified, the status of all groups is changed.

FORCEWRITE
Causes all deferred writes to be written.

FLUSHRO
Releases the read-only blocksets from the cache, sets the timestamp entry to 0, and puts an entry at the end of the most recently used chain.

CLOSE
Working in conjunction with FORCEWRITE, puts entries onto blockset free chain and closes the designated cache file entry.

CLEAR.STAT
Clears the statistics from a specific file or from the global cache.

ALL Releases all semaphores. ALL is mutually exclusive of DCFILE, ENTRY, ARRAY, FREECHAIN, and DAEMON.

DCFILE
Releases the cache file chain semaphore. DCFILE is mutually exclusive of ALL but can be included in combination with ENTRY, ARRAY, FREECHAIN, or DAEMON.

ENTRY
Releases the cache file entry semaphore. ENTRY is mutually exclusive of ALL but can be included in combination with DCFILE, ARRAY, FREECHAIN, or DAEMON.

ARRAY
Releases the cache file array semaphore. ARRAY is mutually exclusive of ALL but can be included in combination with DCFILE, ENTRY, FREECHAIN, or DAEMON.

FREECHAIN
Releases the blockset freechain semaphore. FREECHAIN is mutually exclusive of ALL but can be included in combination with DCFILE, ENTRY, ARRAY, or DAEMON.

DAEMON
Releases the cache daemon semaphore. DAEMON is mutually exclusive of ALL but can be included in combination with DCFILE, ENTRY, ARRAY, or FREECHAIN.

USER *x*
Identifies the pid of the user owning the semaphore to be released. If omitted, all users is assumed. USER can be specified with ALL, DCFILE, ENTRY, ARRAY, FREECHAIN, and DAEMON to limit these options to a specific user.

SEMNO *y*
Specifies the number of array or entry semaphores to be released. ENTRY or ARRAY must also be specified. If omitted, all entry or array semaphores is assumed.

STOP {DAEMON}
Stops the disk cache asynchronous write daemon.

ABORT [DETAIL]
Stops everything, flushes all the files, clears all semaphores and statistics, and stops the daemon. If DETAIL is specified, steps are shown. If ABORT is specified, DETAIL is the only other parameter permitted.

Placing Files Permanently in the Disk Cache

The administrator can place specific files permanently in the disk cache with the following server engine command:

```
CATALOG.FILE.CACHE filename {PRE.LOAD|WRITE.DEFER}
```

The command has the following components:

Components
Description

FILE *filename*
Names the file to be permanently placed in disk cache.

PRE.LOAD
Loads the data of the file into cache memory.

WRITE.DEFER
Defers writing to the file.

The administrator can preload a read-only or write-cached mode file into cache memory. It remains there between normal uses. At a minimum, its modified records are written to disk when the last user closes it while in write-defer mode.

Removing Files from the Disk Cache

The administrator can remove a file from cache memory with the following command:

```
DECATALOG.FILE.CACHE filename
```

The command has the following component:

Components
Description

FILE *filename*
Names the file to be removed from disk cache.

The file is flushed and removed from cache when the last current user closes the file.

Starting and Stopping the Cache Daemon

The administrator can start and stop the asynchronous background cache (writer) daemon.

Note: You must be logged into the dshome account. See "Logging into the dshome Account" for information about logging into the dshome account.

The command is:

```
DAEMON.FILE.CACHE [[START x ] | STOP]
```

The command has the following components:

Components
Description

START
Starts the asynchronous background cache daemon.

x Identifies the pause period between scans. *x* is expressed in 10-millisecond units.

STOP Stops the asynchronous background cache daemon.

Logging into the dshome Account

The CLEAR.FILE.CACHE command and the DAEMON.FILE.CACHE command require that you log in as administrator and be logged into the IBM InfoSphere DataStage home (dshome) account.

In UNIX:

To log in to the InfoSphere DataStage home account:

1. Log in as *dsadm*.
2. Determine the path for *dshome*:
   ```
   cat /.dshome
   ```
3. Change the directory to the specified path. For example, if the path is */u1/uv*, the command is:
   ```
   cd /u1/uv
   ```
4. Log in to the home account
   ```
   bin/dssh
   ```

You are now in the InfoSphere DataStage home account.

In Windows:

To log in to the InfoSphere DataStage home account:

From a command prompt, change to the server engine directory and issue the following command:

```
C:\IBM\InformationServer\Server\DSEngine\bin\dssh
```

where ***C:\IBM\InformationServer\Server\DSEngine*** is the installed server engine location. You are now in the home account.

Tuning Link Public Caching and System Caching

The administrator can use the following tunables in the *uvconfig* file to tune the performance of disk caching.

Tunable
Description

DISKCACHE
Specifies the state of the disk cache subsystem. This tunable must have a positive value when using either link public caching or system caching. The following are the valid values:

- -1, meaning ALLOW. The disk cache is inactive. Files opened in read-only or write-cache mode are processed as if opened in read/write mode. This is the default value.
- 0, meaning REJECT. The disk cache subsystem is inactive. Files opened in read-only or write-cache mode produce an error.
- *n*. The disk cache subsystem is active. *n* represents the size of the disk cache shared memory in megabytes. Values 1 - 512 are allowed. The shared cache is limited to 512 mb on all platforms except Compaq Tru64, which has a limit of 176 mb.

DCBLOCKSIZE
Specifies the size of a shared memory disk cache buffer in 1K units (1024 bytes). Valid values are 4, 8, 16, and 32. 16 is the default value.

DCMODULUS
Specifies the number of chains of shared memory disk cache buffers into which a file is divided. Valid values are 128, 256, 512, and 1024. 256 is the default value. This tunable is specific to system caching.

DCMAXPCT
Specifies the percentage of the total shared memory disk cache buffers that can be owned by a file. Valid values are 1 - 100. 80 is the default value. This tunable is specific to system caching.

DCFLUSHPCT
Specifies the percentage of the total shared memory disk cache buffers owned by a file that can

be in a write-deferred state before they are flushed to disk. Valid values are 1 - 100. 80 is the default value. This tunable is specific to system caching.

DCCATALOGPCT
Specifies the percentage of the total shared memory disk cache buffers that can be owned by data files that are cataloged for disk caching. Valid values are 1 - 100. 50 is the default value. This tunable is specific to system caching.

DCWRITEDAEMON
Specifies the state of the shared memory disk cache background write daemon. The following are the valid values:

- 0 is the default value and indicates the background write daemon is inactive.

n indicates the write daemon is active. *n* is the amount of time the write daemon pauses between writes, expressed in 10-millisecond units. This tunable is specific to system caching.

Using the Euro Symbol on Non-NLS systems

If you want to include the Euro symbol in hashed files on non-NLS systems, you have to take some steps to support the symbol. See "Using the Euro Symbol on Non-NLS systems" on page 42 for information.

Considerations for Performance

Consider the following to improve job performance.

Single versus Multiple Jobs

System caching allows multiple jobs or stages running concurrently to share the same server engine files, either as read only or for writing and updating. System caching is not intended to be used if only a single stage is creating or reading the file.

Write-Deferred Caching

This type of system caching offers the best performance because expensive synchronous writes to the physical disk file are deferred. For demand updating, no separate write cache daemon is active, and updated blocks are only written to disk when a file's blockset quota is exceeded or the last opened reference to the file has been closed. Lazy updating is demand updating augmented by a separate asynchronous writer daemon. The write daemon is not required for deferred updating. However when active, it can help to minimize blockset quota limits and reduce the possibility of file corruption by keeping the file in a more consistent state.

With write-deferred caching, the actual deferred blocks of a blockset that are written to a disk are determined by a least-recently-used aging algorithm. While this option provides the best overall performance, if the server engine crashes, the file might be corrupted.

Write-Immediate Caching

This type of system caching has slower performance than write-deferred caching because writes to the physical disk file are happening at the same time the cache is updated. While this option reduces performance, it avoids file corruption as much as possible should the server engine crash.

Performance Improvements

A set of server engine files can be cached in shared memory segments of a size determined by the *uvconfig* file. If a majority of the referenced groups are in this dynamic cache, performance is improved. If all groups of a file are referenced randomly and do not all fit in the disk cache, performance can be

worse than if no caching is in effect. Also, if the host operating system has less physical memory than the size of the configured disk cache, performance suffers.

The *uvconfig* file in the IBM InfoSphere DataStage home directory has a number of tunables that are used by the disk cache (see "Tuning Link Public Caching and System Caching" on page 327. Any platform can hold a subset of a file or table in cache with aged blocksets released when new blocksets are needed. When a number of large files are in cache, only a small subset of a file's blocks will reside in the cache, but the administrator can modify the tunables to allow a small subset to be handled efficiently. DCFLUSHPCT gives the administrator control over the blockset replacement algorithm to prevent read-only starvation. DCMAXPCT controls the maximum percent of the cache that can be occupied by one file. The disk cache knows when no active process requires access to a file's block and releases it if necessary.

Optimal performance is achieved when the size of the disk cache shared memory, which is set with the tunable DISKCACHE, is set high enough to contain the whole file or to contain a high proportion (90-95%) of the referenced groups. If the DISKCACHE size is inappropriately small, thrashing occurs in the disk cache.

DCMODULUS has some effect on run time, especially for large files. As this number decreases, the length of active chains of DCBLOCKSIZE buffers increases resulting in increased time to execute a sequential search for an entry. Setting DCMODULUS to 1024 generally is optimal for large files (greater than 75 mb). The penalty is that fewer disk cache file structures fit in one cache buffer, thus removing a few from the pool of available buffers.

The default setting of DCBLOCKSIZE is 16. Making it smaller results in increased physical I/O and array chains with increasing length, both slowing down the system. DCBLOCKSIZE should be made larger than 16 only if the platform can handle the extended physical I/O requests in one I/O to its disk subsystem. One way to recognize this is if the platform has disk arrays.

Disk cache blocks are stored in *n* block sets (where *n* is configured as 4k, 8k, 16k or 32k with 16k as the default) to reduce sequential search time and allow prereading of blocks in the area of the one requested. For this reason, file separation size is restricted to a power of 1024 bytes. Each file will own *m* blockset chains (where *m* is configured as 64, 128, 256, 512, or 1024 with 256 as the default).

The following are examples of tunable settings such that a single file is held in memory with acceptable array-referenced blockset chain lengths that must be scanned sequentially to find a block.

Table 19. Example settings

File Size	DCBLOCKSIZE	DCMODULUS	Average Blockset Chain Length
32mb	8k	128	32
32mb	8k	256	16
64mb	8k	256	32
64mb	16k	256	16
160mb	16k	256	40
320mb	16k	512	40
1024mb	16k	256	256
1024mb	16k	512	128
1024mb	16k	1024	64
1024mb	32k	1024	32
2048mb	32k	1024	64

Type 30 files are really two files: one for primary groups and the other for overflow blocks. Therefore, files both must be considered when setting DCMAXPCT.

The value of DCWRITEDAEMON determines the amount of time the write daemon pauses between writes. On a multiprocessor platform, the write daemon pause period, which is specified in DCWRITEDAEMON, can be set quite low; on a single-processor the value should be 10 or greater.

Product accessibility

You can get information about the accessibility status of IBM products.

The IBM InfoSphere Information Server product modules and user interfaces are not fully accessible. The installation program installs the following product modules and components:

- IBM InfoSphere Business Glossary
- IBM InfoSphere Business Glossary Anywhere
- IBM InfoSphere DataStage
- IBM InfoSphere FastTrack
- IBM InfoSphere Information Analyzer
- IBM InfoSphere Information Services Director
- IBM InfoSphere Metadata Workbench
- IBM InfoSphere QualityStage

For information about the accessibility status of IBM products, see the IBM product accessibility information at http://www.ibm.com/able/product_accessibility/index.html.

Accessible documentation

Accessible documentation for InfoSphere Information Server products is provided in an information center. The information center presents the documentation in XHTML 1.0 format, which is viewable in most Web browsers. XHTML allows you to set display preferences in your browser. It also allows you to use screen readers and other assistive technologies to access the documentation.

IBM and accessibility

See the IBM Human Ability and Accessibility Center for more information about the commitment that IBM has to accessibility.

Accessing product documentation

Documentation is provided in a variety of locations and formats, including in help that is opened directly from the product client interfaces, in a suite-wide information center, and in PDF file books.

The information center is installed as a common service with IBM InfoSphere Information Server. The information center contains help for most of the product interfaces, as well as complete documentation for all the product modules in the suite. You can open the information center from the installed product or from a Web browser.

Accessing the information center

You can use the following methods to open the installed information center.

- Click the **Help** link in the upper right of the client interface.

  Note: From IBM InfoSphere FastTrack and IBM InfoSphere Information Server Manager, the main Help item opens a local help system. Choose **Help > Open Info Center** to open the full suite information center.
- Press the F1 key. The F1 key typically opens the topic that describes the current context of the client interface.

  Note: The F1 key does not work in Web clients.
- Use a Web browser to access the installed information center even when you are not logged in to the product. Enter the following address in a Web browser: http://host_name:port_number/infocenter/topic/com.ibm.swg.im.iis.productization.iisinfsv.home.doc/ic-homepage.html. The host_name is the name of the services tier computer where the information center is installed, and port_number is the port number for InfoSphere Information Server. The default port number is 9080. For example, on a Microsoft® Windows® Server computer named iisdocs2, the Web address is in the following format: http://iisdocs2:9080/infocenter/topic/com.ibm.swg.im.iis.productization.iisinfsv.nav.doc/dochome/iisinfsrv_home.html.

A subset of the information center is also available on the IBM Web site and periodically refreshed at http://publib.boulder.ibm.com/infocenter/iisinfsv/v8r5/index.jsp.

Obtaining PDF and hardcopy documentation

- PDF file books are available through the InfoSphere Information Server software installer and the distribution media. A subset of the PDF file books is also available online and periodically refreshed at www.ibm.com/support/docview.wss?rs=14&uid=swg27016910.
- You can also order IBM publications in hardcopy format online or through your local IBM representative. To order publications online, go to the IBM Publications Center at http://www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss.

Providing feedback about the documentation

You can send your comments about documentation in the following ways:

- Online reader comment form: www.ibm.com/software/data/rcf/
- E-mail: comments@us.ibm.com

Links to non-IBM Web sites

This information center may provide links or references to non-IBM Web sites and resources.

IBM makes no representations, warranties, or other commitments whatsoever about any non-IBM Web sites or third-party resources (including any Lenovo Web site) that may be referenced, accessible from, or linked to any IBM site. A link to a non-IBM Web site does not mean that IBM endorses the content or use of such Web site or its owner. In addition, IBM is not a party to or responsible for any transactions you may enter into with third parties, even if you learn of such parties (or use a link to such parties) from an IBM site. Accordingly, you acknowledge and agree that IBM is not responsible for the availability of such external sites or resources, and is not responsible or liable for any content, services, products or other materials on or available from those sites or resources.

When you access a non-IBM Web site, even one that may contain the IBM-logo, please understand that it is independent from IBM, and that IBM does not control the content on that Web site. It is up to you to take precautions to protect yourself from viruses, worms, trojan horses, and other potentially destructive programs, and to protect your information as you deem appropriate.

Notices and trademarks

This information was developed for products and services offered in the U.S.A.

Notices

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785 U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003 U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at www.ibm.com/legal/copytrade.shtml.

The following terms are trademarks or registered trademarks of other companies:

Adobe is a registered trademark of Adobe Systems Incorporated in the United States, and/or other countries.

IT Infrastructure Library is a registered trademark of the Central Computer and Telecommunications Agency which is now part of the Office of Government Commerce.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

ITIL is a registered trademark, and a registered community trademark of the Office of Government Commerce, and is registered in the U.S. Patent and Trademark Office

UNIX is a registered trademark of The Open Group in the United States and other countries.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

The United States Postal Service owns the following trademarks: CASS, CASS Certified, DPV, LACS^{Link}, ZIP, ZIP + 4, ZIP Code, Post Office, Postal Service, USPS and United States Postal Service. IBM Corporation is a non-exclusive DPV and LACS^{Link} licensee of the United States Postal Service.

Other company, product or service names may be trademarks or service marks of others.

Contacting IBM

You can contact IBM for customer support, software services, product information, and general information. You also can provide feedback to IBM about products and documentation.

The following table lists resources for customer support, software services, training, and product and solutions information.

Table 20. IBM resources

Resource	Description and location
IBM Support Portal	You can customize support information by choosing the products and the topics that interest you at www.ibm.com/support/entry/portal/Software/Information_Management/InfoSphere_Information_Server
Software services	You can find information about software, IT, and business consulting services, on the solutions site at www.ibm.com/businesssolutions/
My IBM	You can manage links to IBM Web sites and information that meet your specific technical support needs by creating an account on the My IBM site at www.ibm.com/account/
Training and certification	You can learn about technical training and education services designed for individuals, companies, and public organizations to acquire, maintain, and optimize their IT skills at http://www.ibm.com/software/sw-training/
IBM representatives	You can contact an IBM representative to learn about solutions at www.ibm.com/connect/ibm/us/en/

Providing feedback

The following table describes how to provide feedback to IBM about products and product documentation.

Table 21. Providing feedback to IBM

Type of feedback	Action
Product feedback	You can provide general product feedback through the Consumability Survey at www.ibm.com/software/data/info/consumability-survey
Documentation feedback	To comment on the information center, click the Feedback link on the top right side of any topic in the information center. You can also send comments about PDF file books, the information center, or any other documentation in the following ways: • Online reader comment form: www.ibm.com/software/data/rcf/ • E-mail: comments@us.ibm.com

Index

Special characters

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Printed in USA

SC18-9898-03

Spine information:

IBM InfoSphere DataStage **Version 8 Release 5** **Server Job Developer's Guide**