Net.Data
Language Environment Reference

Net.Data
Language Environment Reference

Note

Be sure to read the information in “Appendix B. Notices” on page 85 before using this information and the product it supports.

Fourth Edition (June 1998)

Contents

Preface v
About Net.Data v
About This Book v
  Who Should Read This Book vi
  About Examples in This Book vi

About Net.Data Language Environments vii

Part 1. Net.Data-Supplied Language Environments 1

Chapter 1. Overview of Net.Data Supplied Language Environments . . . 3

Chapter 2. Using the Net.Data-Supplied Language Environments 5
Flat File Interface Language Environment. 5
  Security Considerations 5
  The FFI Built-in Functions 6
IMS Web Language Environment. 8
Java Applet Language Environment 9
  Creating Java Applets 9
  Generating the Applet Tags 9
  Java Applet Example 13
  Using the Net.Data Java Applet Interface 14
Java Application Language Environment 15
  Java Language Environment File Structure 16
  Creating the Java Function 16
  Defining the Java Language Environment Cliette 17
  Configuring Net.Data for the Java Language Environment. 17
  Creating and Running the Macro File 18
ODBC Language Environment 18
Oracle Language Environment. 19
Perl Language Environment. 21
REXX Language Environment 22
SQL Language Environment 23
Sybase Language Environment 24
System Language Environment 26
Web Registry Language Environment 26
Configuring Net.Data Language Environments 28

Part 2. Non-IBM Language Enviroments. 29

Chapter 3. Creating a New Language Environment 31
Designing a DLL or Shared Library 31
  Which Language Environment Interfaces Should I Provide? 32
  Processing Input Parameters 32
  Processing User Requests 33
  Processing Output Parameters. 33
  Communicating Error Conditions 33
Language Environment Communication Structures 33
  The dtw_lei Structure 34
  The dtw_parm_data Structure 35
Language Environment Interface Functions 36
  dtw_initialize() 37

dtw_execute() 38
dtw_getNextRow() 38
dtw_cleanup() 39
Designing the Language Environment Statement 39
ENVIRONMENT Statement Syntax 40
ENVIRONMENT Statement Examples 40

Chapter 4. The Language Environment Programming Interface Utility Functions 43
Language Environment Utility Functions 43
Utility Functions for Managing Memory. 43
Utility Functions for Managing Configuration Variables 43
Utility Functions for Table Manipulation. 44
Utility Functions for Row Manipulation 45
Utility Functions Syntax Reference 45
dtw_free() 46
dtw_getvar() 47
dtw_malloc() 48
dtw_row_SetCols() 49
dtw_row_SetV() 50
dtw_strdup() 51
dtw_table_AppendRow() 52
dtw_table_Cols() 53
dtw_table_Delete() 54
dtw_table_DeleteCol() 55
dtw_table_DeleteRow() 56
dtw_table_GetN() 57
dtw_table_GetV(). 58
dtw_table_InsertCol() 59
dtw_table_InsertRow() 60
dtw_table_MaxRows() 61
dtw_table_New() 62
dtw_table_QueryColnoNj() 63
dtw_table_Rows() 64
dtw_table_SetCols(). 65
dtw_table_SetN(). 66
dtw_table_SetV(). 67

Part 3. Appendixes 69

Appendix A. Language Environment Template 71

Appendix B. Notices 85
Trademarks. 86

Glossary 87

Index 89

Preface

Thank you for selecting Net.Data Version 2, IBM’s development tools for creating dynamic Web pages! With Net.Data you can rapidly develop Web pages with a dynamic content by incorporating data from a variety of data sources and by using the power of programming languages you already know.

Net.Data Version 2 provides significantly improved performance along with new features that give you the power to build and deploy your Internet business solutions.

About Net.Data

With IBM’s Net.Data product, you can create dynamic Web pages using data from both relational and non-relational database management systems (DBMSs), including DB2, IMS, and ODBC-enabled databases, and using applications written in programming languages such as Java, JavaScript, Perl, C, C++, and REXX.

You can think of Net.Data as a macro processor that executes as middleware on a Web server. You can write Net.Data application programs, called macros, that Net.Data interprets to create dynamic Web pages with customized content based on input from the user, the current state of your databases, existing business logic, and other factors that you design into your macro.

A request, in the form of a URL (uniform resource locator), flows from a browser, such as Netscape or Internet Explorer, to a Web server that forwards the request to Net.Data for execution. Net.Data locates and executes the macro, and builds a Web page that it customizes based on functions that you write. These functions can:

- Encapsulate business logic within Perl scripts, C and C++ applications, or REXX programs
- Access databases such as DB2

Net.Data supports industry-standard interfaces such as HyperText Transfer Protocol (HTTP) and Common Gateway Interface (CGI). HTTP is used between the browser and the Web server, and CGI is used between the Web server and Net.Data. This lets you select your favorite browser or web server for use with Net.Data. Net.Data also supports FastCGI and the major Web server APIs on multiple operating systems.

About This Book

This book discusses Net.Data’s language environments, which are used when you call programs or functions, or data sources like DB2, Oracle, or Sybase databases from your Net.Data macro file. It describes each language environment supplied with Net.Data, as well as describes the language environment interface that you can use to design and build your own language environment.

This book might refer to products or features that are announced, but not yet available.

More information including sample Net.Data macros, demos, and the latest copy of this book, is available from the following World Wide Web sites:

- http://www.software.ibm.com/data/net.data

- http://www.as400.ibm.com/netdata

Who Should Read This Book

People writing Net.Data macros can use this information to learn about the capabilities of the language environments that Net.Data provides. This book also contains information for people who want to write their own language environments for Net.Data.

To understand the concepts discussed in this book, you should be familiar with the C programming language and the information in *Net.Data Administration and Programming Guide* and *Net.Data Reference*.

About Examples in This Book

Examples used in this book are kept simple to illustrate specific concepts and do not consider every possible case. Some examples are fragments that do not work alone.

About Net.Data Language Environments

Net.Data is designed to allow new programming language and database interfaces to be added in a *pluggable* fashion. These interfaces are called language environments and are accessed as DLLs or shared libraries. Language environments provide access to applications and databases that support your dynamic Web pages. By invoking language environments with function calls and SQL statements, you can access the functions and utilities that these language environments provide for use with your business application. For example, you can directly access your ODBC database, use the Perl language environment to call Perl scripts, or call the Java Applets language environment to run Java applets.

The Net.Data initialization file associates each language environment name with a DLL or shared library. Each language environment must support a standard set of interfaces defined by Net.Data. Net.Data loads the DLL or shared library specified in the initialization file the first time that a function call for a FUNCTION block specifying that language environment is encountered.

Net.Data parses the Net.Data macro, maintains the Net.Data variables, communicates with the language environments, and formats the output according to the REPORT and MESSAGE block specifications. The language environment supports the interfaces defined to Net.Data, makes the Net.Data parameters accessible to the language processor in some language-dependent manner, calls the language interpreter, and receives the variables back from the language interpreter in some language-dependent manner.

Figure 1 demonstrates Net.Data's interaction with language environments.

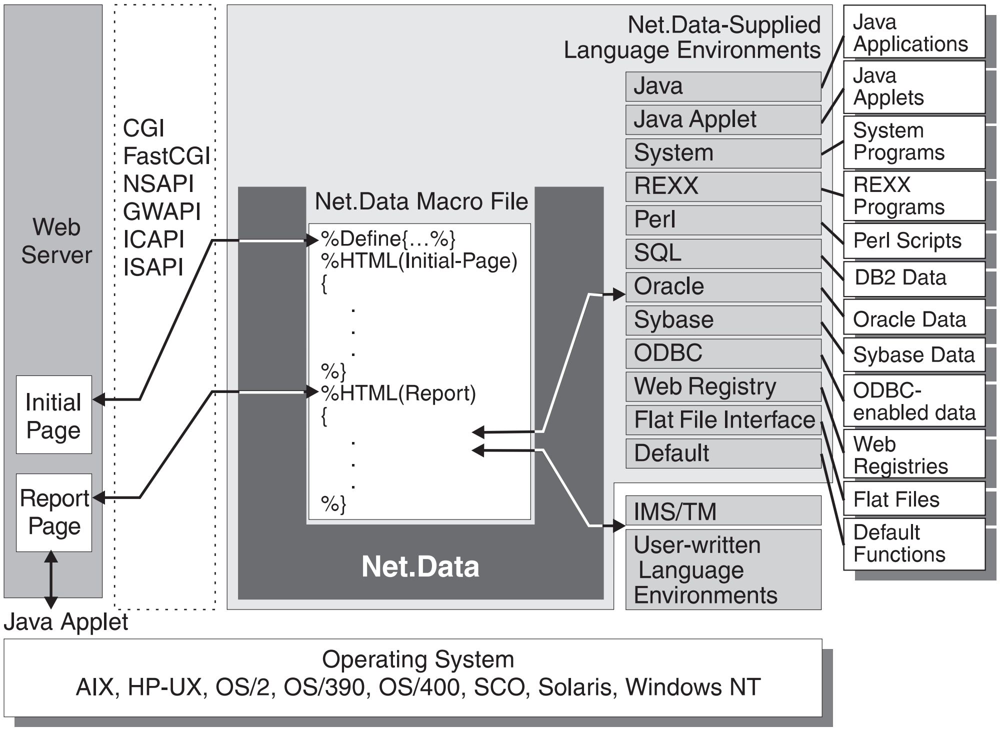

Figure 1. Net.Data and Language Environments

Working with language environments in a Net.Data application involves two kinds of tasks.

- Using the Net.Data-supplied language environments to develop a Net.Data application.
- Developing new language or database environments for other users to use when developing Net.Data applications.

This book is organized to help you with both tasks:

- “Part 1. Net.Data-Supplied Language Environments” on page 1 describes the Net.Data supplied language environments that you can use with your Net.Data applications.
- “Part 2. Non-IBM Language Enviroments” on page 29 describes how to create new language and database environments.

Part 1. Net.Data-Supplied Language Environments

Net.Data supplies several language environments that let you pass information to and from data sources. For example, the language environment for SQL lets you pass native SQL queries to a database. Likewise, the REXX language environment lets you invoke REXX programs.

This section describes each language environment and how to configure language environments in the Net.Data initialization file.

“Chapter 1. Overview of Net.Data Supplied Language Environments” on page 3

“Flat File Interface Language Environment” on page 5

“IMS Web Language Environment” on page 8

“Java Applet Language Environment” on page 9

“Java Application Language Environment” on page 15

“ODBC Language Environment” on page 18

“Oracle Language Environment” on page 19

“Perl Language Environment” on page 21

“REXX Language Environment” on page 22

“SQL Language Environment” on page 23

“Sybase Language Environment” on page 24

“System Language Environment” on page 26

“Web Registry Language Environment” on page 26

“Configuring Net.Data Language Environments” on page 28

Chapter 1. Overview of Net.Data Supplied Language Environments

Net.Data provides a number of language environments, although some operating systems do not support all environments. Table 1 lists the IBM-supplied language environments. To determine whether a language environment is supported on your operating systems, see the operating system reference appendix of *Net.Data Reference*. See your Net.Data README file or Program Directory for details about the language environment statements your operating system uses.

Table 1. Net.Data Language Environments

Language Environment	Environment Statement	Description
Flat File Interface	DTW_FILE	The flat file interface (FFI) provides functions that support text files as data sources.
IMS Web	HWS_LE	The IMS Web language environment lets you submit an IMS transaction using IMS Web and receive the output of the transaction at your Web browser.
Java Applet	DTW_APPLET	The Java applet language environment lets you use Java applets in your Net.Data applications. To generate an applet tag, you must provide the applet tag's qualifiers and the applet's parameter list.
Java Application	DTW_JAVAPPS	Net.Data supports your existing Java applications with the Java language environment.
ODBC	DTW_ODBC	The ODBC language environment executes SQL statements through an ODBC interface for access to multiple database management systems.
Oracle	DTW_ORA	The Oracle language environment lets you directly access your Oracle data.
Perl	DTW_PERL	The Perl language environment interprets internal Perl scripts that are specified in a FUNCTION block of the Net.Data macro, or it executes external Perl scripts stored in separate files.
REXX	DTW_REXX	The REXX language environment interprets internal REXX programs that are specified in a FUNCTION block of the Net.Data macro, or it can execute external REXX programs stored in a separate file.
SQL	DTW_SQL	The SQL language environment executes SQL statements through DB2. The results of the SQL statement can be returned in a table variable.
Sybase	DTW_SYB	The Sybase language environment lets you directly access your Sybase data.
System	DTW_SYSTEM	The System language environment supports calls to external programs that are identified in an EXEC statement in the FUNCTION block. The System language environment interprets the EXEC statement by passing the program name and its parameters to the operating system for execution.
Web Registry	DTW_WEBREG	The Web Registry language environment provides functions for the persistent storage of application-related data.

Each language environment requires an ENVIRONMENT statement in the initialization file and a shared library or DLL file in the server's `/lib` or `/dll` directory. See the configuring chapter in *Net.Data Administration and Programming Guide* for more information.

Recommendation: Make a backup of your initialization file before making changes.

Chapter 2. Using the Net.Data-Supplied Language Environments

The following sections describe the Net.Data-supplied language environments, as well as steps for setting up and using them.

Flat File Interface Language Environment

If you choose to use flat files (or plain-text files) as your data source, use the flat file interface (FFI) and its associated functions to open, close, read, write, and delete files on the Web server. You must specify a path for the FFI_PATH variable in the initialization file.

The file language support uses FFI functions to read from or write to files on the Web server at the Web client's request through the browser. FFI views the file as a record file, each record equivalent to a row in a Net.Data macro table variable, and each value in a record equivalent to a field value in a Net.Data macro table variable. FFI reads records from a file into rows of a Net.Data macro table, and writes rows from a table into records.

Security Considerations

You can specify which files FFI functions can access with the FFI_PATH statement in the Net.Data initialization file. FFI only searches the paths listed in the statement, so files in other directories are safe. This is an example statement:

```
FFI_PATH     C:\public;.\;E:\WWW;E:\guest;A:
```

The paths listed in FFI_PATH are searched from first to last. Net.Data uses the first copy that it finds. If the FFI_PATH is not in the initialization file, FFI attempts to find the file in the current directory. The Net.Data initialization file is shipped without FFI_PATH.

Recommendations:

- Choose which directories are appropriate to use for flat file operations. These directories need to be added to the FFI_PATH to limit searching to those directories.
- Use care letting people perform DTWF_REMOVE or other export operations in the macro to prevent people from removing or altering files with extensions `.dll` and `.cmd` that you might have in the current directory.
- Take appropriate steps to safeguard the files on the system by using reasonable control over what macros are added to the system.
- Do not specify a path in FFI_PATH that lets anonymous FTP users write to the path. If you do, somebody can put a Net.Data macro on the system that allows actions that were not previously allowed.
- Do not add the path of the Net.Data initialization file to the FFI_PATH.

Authorization Tip: Ensure that the Web server has access rights to files used by the FFI built-in functions. See the section on specifying Web server access rights to Net.Data files in the configuration chapter of *Net.Data Administration and Programming Guide* for more information.

The FFI Built-in Functions

This section describes usage tips and issues to consider when using the FFI built-in functions

General considerations

- You can import any plain-text file, but Net.Data macro syntax is interpreted by Net.Data and HTML tags that might be in the text are used to format the text by the browser.
- The FFI parameters are case sensitive only if the operating system is case sensitive.

Current directory

- The current directory for Net.Data depends on the configuration of your Web server. If you are using CGI, the current directory is the directory that Net.Data is running from, which is normally `\www\cgi-bin`. If you are using a Web server API, the current directory can vary. To write to the current directory, Net.Data (or the user ID associated with thread or process that is executing Net.Data) must have write permission. If the server's default request routing or resource mapping is changed, the current directory might be changed also.
- The recommended way to specify the current directory is to use absolute paths for the FFI_PATH statement and for the FILENAME parameter, especially if you are using a Web server API. All directories and sub-directories listed in the path must be defined in the FFI_PATH path statement in the initialization file, or Net.Data will not be able to find the file. For example, the following path requires that the directory and the sub-directories `/u/user/mydir/` be listed in the FFI_PATH.

  ```
  filename="/u/user/mydir/myfile.txt"
  ```

  If you specify just the file name, as in the following example.

  ```
  filename="myfile.txt"
  ```

  Then Net.Data concatenates all the directory names in the path for FFI_PATH and searches for the file first. If it cannot find the file then, Net.Data assumes the file is in the current directory. If the file is not in the current directory, a file with the specified file name will be created in the current directory. If Net.Data does not have write permission to the current directory, an error occurs. Do not use the following syntax:

  ```
  filename="/myfile.txt"
  ```

DELIMITER parameter

- The delimiter is the flag or separator that FFI uses when dividing the file into parts (such as columns of a row) according to the requested transform.
- For read operations, the delimiter separates the contents of the file into rows and columns of a table. For write operations, the delimiter indicates the end of a value in a table row and column.

  Net.Data passes the delimiter to the FFI as a Net.Data macro string and does not include a null character at the end of the characters unless explicitly listed in the DELIMITER parameter. To use the null character in the delimiter, specify the DELIMITER parameter a back slash and a zero in double quotes, "/0", instead of an empty string by using two double quotes, """". If you specify the ASCIITEXT transform, Net.Data uses the new-line character as the delimiter and ignores any requested delimiter.

- Undesirable changes to a file can occur if you use a different delimiter for write operations than for read operations. If you use a different delimiter for a write operation than you use for the read operation, Net.Data writes the file with the new delimiter.
- The maximum length of a delimiter is 256 characters.

FFI_PATH

- Paths in FFI_PATH must contain valid printable characters. FFI does not allow paths that include a question mark (?) or double quotes (“”).
- Sub-directories of the paths listed in FILENAME are not searched unless explicitly specified in FFI_PATH. Specify all directories and sub-directories in the FFI_PATH that you use with the *filename* parameter in the macro file. The following examples show recommended path statements:

  Example 1: Specifies an absolute path listing all directories and sub-directories

  ```
  filename="/u/usr/mydir/myfile.txt"
  ```

  Net.Data searches the allowed paths in the FFI_PATH; if the absolute path assigned to the FILENAME parameter is wrong or not available, Net.Data searches the current directory and if it does not fine the file name, issues a warning.

  Example 2: Specifies a file name in the current directory

  ```
  filename="myfile".txt
  ```

  Net.Data creates new files in the current directory. If Net.Data does not have permission to create files in the directory, Net.Data (or the user ID associated with the thread or process that is executing Net.Data) issues a warning.

DTWF_SEARCH function

- The table returned for DTWF_SEARCH has three columns. The first two columns contain the row and the column number where the match was found; the last column contains the column value that contains the characters specified in the *SearchFor* parameter. For example, if the fourth row of the file contains matching characters in column three, the returned table has a row with the number `4` in the first column to indicate the row of the file that it came from; it has a number `3` in the second column to indicate which column of the file contains a match; and it has the complete column value in the third column.
- The *SearchFor* parameter cannot include the contents of the delimiter parameter.

STARTROW and ROWS parameters

- For the functions DTWF_DELETE, DTWF_INSERT, DTWF_UPDATE, and DTWF_WRITE, if a *StartRow* value larger than the last row is specified, *StartRow* is changed to indicate the last row and an error is returned.
- For the functions DTWF_READ and DTWF_SEARCH, the *Rows* value is returned as the number of rows in the table.

TABLE parameter

- The maximum length of a row in an FFI table is 16383 characters. This limit includes a null character for each column in the Net.Data macro table.

TRANSFORM parameter

- This parameter indicates how the file is divided into parts with respect to the rows and columns of a Net.Data macro table. For example, ASCIITEXT transform means that each line of the file corresponds with a row of a Net.Data macro table and the Net.Data macro table has only one column. DELIMITED transform means that the characters in the row are examined to find the DELIMITER and after the DELIMITER is the content of the next column.

- A new-line character in a file indicates the end of a row of a Net.Data macro table for ASCIITEXT and DELIMITED transforms.

File locking

- Files are not locked unless you open them with DTWF_OPEN. If a file is not locked, it can change between the time it is read and updated. This can result in the loss of the previous changes. Using DTWF_OPEN opens the file during the execution of the macro using the file system's locking mechanism.

DTWF_APPEND

- The current contents of a file affect the results of using DTWF_APPEND, especially the contents of the last column of the last row. If a new line follows the last column value of the last row of the file, appended data is placed in a new row. Otherwise, appended data becomes part of the last row of the file.

IMS Web Language Environment

The IMS Web language environment is part of a complete end-to-end solution for running your IMS transactions in the World Wide Web environment. The IMS Web language environment provides:

- A Net.Data macro with:
  - The HTML used to enter the transaction input data
  - A Net.Data FUNCTION block that invokes the IMS Web language environment
  - The HTML that displays the output of the transaction
- A transaction DLL or shared library that is invoked by the IMS Web language environment

Restriction: The IMS Web language environment of Net.Data is only supported when Net.Data runs as a CGI application. It is not supported by Net.Data with ICAPI.

The IMS Web Studio tool generates code for the DLL and the macro, as well as a MAK file for building the DLL or shared library, from the (Message Format Service) MFS source for the transaction. After the executable form of the DLL has been built, the DLL and the macro files are moved to the Web server that is running Net.Data. The transaction is ready to run in the Web environment.

To use the IMS Web language environment:

1. Install the IMS Web Runtime component on the Web server running Net.Data. For information about the IMS Web Runtime component, see *IMS Web User's Guide*:

   `http://www.software.ibm.com/data/ims/about/imsweb/document/index.html`
2. Create the transaction DLL file.
   a. Generate the C++, MAK, and Net.Data macro files for your transaction with the IMS Web Studio tool.
   b. If you are running Net.Data on an operating system that is different than the operating system on which the IMS Web Studio tool is run, move the DLL source files to an IMS Web development machine for the target operating

system. For example, if you run the IMS Web Studio tool on Windows NT and the target platform is AIX or OS/390, move the source for the transaction DLL to an IMS Web development machine running under AIX or OS/390, respectively.

   c. Build the executable form of the transaction DLL using the generated MAK file.

3. Copy the transaction DLL file (DTW*proj*.dll) and Net.Data macro file (DTW*proj*.d2w) to your Web server.
   a. Place the macro in a directory from which Net.Data retrieves macros. (See the MACRO_PATH statement in the configuring chapter of *Net.Data Administration and Programming Guide.)*
   b. Place the transaction DLL or shared library in a directory from which the Web server retrieves DLLs or shared libraries.
4. Use the link in the sample file that is generated by the IMS Web Studio tool, DTW*proj*.htm, to modify an HTML file in your Web server's HTML tree. You can then use the link to invoke Net.Data and display the input HTML form for the transaction on a Web browser. Fill in the transaction input, and select the SUBMIT push button on the form to run the transaction and receive its output at the Web browser.

IMS Web uses the IMS TCP/IP Open Transaction Manager Access (OTMA) Connection to communicate between the Web server and IMS environments. See the IMS Web home page for more information:

`http://www.software.ibm.com/data/ims/about/imsweb`

Java Applet Language Environment

The Java applet language environment lets you easily generate HTML tags for Java applets in your Net.Data applications. When you call the Java applet language environment, you specify the name of your applet and pass any parameters that the applet needs. The language environment processes the macro and generates the HTML applet tags, which the Web browser uses to run the applet.

Additionally, Net.Data provides a set of interfaces your applet can use to access table parameters. These interfaces are contained in the class, DTW_Applet.class.

The following sections describe how to use the Java applet language environment to run your Java applets.

Creating Java Applets

Before using the Net.Data Java applet language environment, you need to determine which applets you plan to use or which applets you need to write. See your Java documentation for more information on creating applets.

Generating the Applet Tags

You specify a call to the applet language environment with a Net.Data function call. No declaration is needed for the function call. The syntax for the function call is shown here:

```
@DTWA_AppletName(parm1, parm2, ..., parmN)
```

- DTWA_ identifies the function call to the applet language environment.
- AppletName is the name of the applet for which tags are generated.

- parm1 through parmN are parameters used to generate PARAM tags.

To write a macro file that generates applet tags:

1. Define any parameters required by the applet in the DEFINE section of the macro file. These parameters include any applet tag attributes, Net.Data variables, and Net.Data table parameters that you need as input for the applet. For example:

   ```
   %define{
   DATABASE = "celdial"                 <=Net.Data variable: name of the database
   MyGraph.codebase = "/netdata-java" <=Required applet parameter
   MyGraph.height = "200"               <=Required applet parameter
   MyGraph.width = "400"                <=Required applet parameter
   MyTitle = "This is my Title"         <=Net.Data variable: name of the Web page
   MyTable = %TABLE(all)                <=Table to store query results
   %}
   ```

2. Optional: Specify a query to the database to generate a result set as input for the applet. This is useful when you are using an applet that generates a chart or table. For example:

   ```
   %FUNCTION(DTW_SQL) mySQL(OUT table){
   select name, age from ibmuser.guests
   %}
   ```

3. Specify the function call in the Net.Data macro to call the Java applet language environment and invoke the applet. The function call specifies the name of the applet and the parameters you want to pass to the language environment. These parameters include any Net.Data variables, and Net.Data table or column parameters that you need as input for the applet.

   For example:

   ```
   %HTML(report){                                   <=The start of the HTML block
   @mySQL(MyTable)                                  <=A call to the SQL function
                                                           mySQL
   @DTWA_MyGraph( MyTitle, DTW_COLUMN(ages) MyTable) <=Applet function call
   %}
   ```

4. Invoke Net.Data and run the macro file. See *Net.Data Administration and Programming Guide* to learn how to invoke Net.Data.

Applet Tag Attributes

You can specify attributes for applet tags anywhere in your Net.Data macro. Net.Data substitutes all variables that have the form *AppletName.attribute* into the applet tag as attributes. The syntax for defining an attribute on an applet tag is shown here:

```
%define AppletName.attribute = "value"
```

These attributes are required for all applets:

- *codebase:* The location of the applet, which is identified by a URL.
- *height:* The height of the applet in pixels.
- *width:* The width of the applet in pixels.

For example, if your applet is called MyGraph, you can define these required attributes as shown here:

```
%DEFINE{
MyGraph.codebase = "/netdata-java/"
MyGraph.height   = "200"
MyGraph.width    = "400"
%}
```

The actual assignment need not be in a DEFINE section. You can set the value with the DTW_ASSIGN function. If you do not define a variable for *AppletName.code* variable, Net.Data adds a default *code* parameter to the applet tag. The value of the *code*parameter is *AppletName.class*, where *AppletName* is the name of your applet.

Applet Tag Parameters

You define a list of parameters to pass to the Java applet language environment in the function call. You can pass parameters that include:

- Net.Data variables (including LIST variables)
- Net.Data tables
- Columns of Net.Data tables

When you pass a parameter, Net.Data creates a Java applet PARAM tag in the HTML output with the name and value that you assign to the parameter. You cannot pass string literals or results of function calls.

Net.Data Variable Parameters: You can use Net.Data variables as parameters. If you define a variable in the DEFINE block of the macro and pass the variable value in the DTWA_*AppletName* function call, Net.Data generates a PARAM tag that has the same name and value as the variable. For example, given the following macro statement:

```
%define{

...

MyTitle = "This is my Title"
%}

%HTML(report){
@DTWA_MyGraph( MyTitle, ...)
%}
```

Net.Data produces the following applet PARAM tag:

```
<param name = 'MyTitle' value = "This is my Title" >
```

Net.Data Table Parameters:

Net.Data automatically generates a PARAM tag with the name DTW_NUMBER_OF_TABLES every time the Java applet language environment is called, specifying whether the function call has passed any table variables. The value is the number of table variables that Net.Data uses in the function. If no table variables are specified in the function call, the following tag is generated:

```
<param name = "DTW_NUMBER_OF_TABLES" value = "0" >
```

You can pass one or more Net.Data table variables as parameters on the function call. If you specify a Net.Data table variable on a DTWA_*AppletName* function call, Net.Data generates the following PARAM tags:

Table name parameter tag:

This tag specifies the names of the tables to pass. The tag has the following syntax:

```
<param name = 'DTW_TABLE_i_NAME' value = "tname" >
```

Where `i` is the number of the table based on the ordering of the function call, and *tname* is the name of the table.

Row and column specification parameter tags:

PARAM tags are generated to specify the number of rows and columns a particular table. This tag has the following syntax:

```
<param name = 'DTW_tname_NUMBER_OF_ROWS' value = "rows" >
<param name = 'DTW_tname_NUMBER_OF_COLUMNS' value = "cols" >
```

Where the name of the table is *tname*, *rows* is the number of rows in the table, and *cols* is the number of columns in the table. This pair of tags is generated for each unique table specified in the function call.

Column value parameter tags:

This PARAM tag specifies the column name of a particular column. This tag has the following syntax:

```
<param name = 'DTW_tname_COLUMN_NAME_j' value = "cname" >
```

Where the table name is *tname*, *j* is the column number, and *cname* is the name of the column in the table.

Row value parameter tags:

This PARAM tag specifies the values at a particular row and column. This tag has the following syntax:

```
<param name = 'DTW_tname_cname_VALUE_k' value = "val" >
```

Where the table name is *tname*, *cname* is the column name, *k* is the row number, and *val* is the value that matches the value in the corresponding row and column.

Table Column Parameters: You can pass a table column as a parameter on a function call to generate tags for a specific column. Net.Data generates the corresponding applet tags only for the specified column. A table column parameter uses the following syntax:

```
@DTWA_AppletName(DTW_COLUMN( x )Table)
```

Where *x* is the name or number of the column in the table.

Table column parameters use the same applet tags defined for the table parameters.

Alternate Text for the Applet Tag on Browsers that are not Java-Enabled

The variable DTW_APPLET_ALTTEXT specifies the text to display on browsers that do no support Java or have turned Java support off. For example, the following variable definition:

```
%define DTW_APPLET_ALTTEXT = "<P>Sorry, your browser is not Java-enabled."
```

produces the following HTML tag and text:

```
<P>Sorry, your browser is not Java-enabled.<BR>
```

If this variable is not defined, no alternate text is displayed.

Java Applet Example

The following example demonstrates a Net.Data macro file that calls the Java applet language environment and the resulting applet tag that the language environment generates.

The Net.Data macro file contains the following function calls to the Java applet language environment:

```
%define{
DATABASE = "celdial"
DTW_APPLET_ALTTEXT = "<P>Sorry, your browser is not Java-enabled."
DTW_DEFAULT_REPORT = "no"
MyGraph.codebase = "/netdata-java/"
MyGraph.height = "200"
MyGraph.width = "400"
MyTitle = "This is my Title"
%}
%FUNCTION(DTW_SQL) mySQL(OUT table){
select name, age from ibmuser.guests
%}
%HTML(report){
@mySQL(MyTable)
@DTWA_MyGraph(MyTitle, DTW_COLUMN(ages) MyTable)
%}
```

The Net.Data macro lines in the DEFINE section specify the first line of the applet tag:

```
MyGraph.codebase = "/netdata-java/"
MyGraph.height = "200"
MyGraph.width = "400"
```

The language environment generates an applet tag with the following qualifiers:

```
<applet code = 'MyGraph.class'
codebase = '/netdata-java/' width = '400'
height = '200' >
```

Net.Data returns the SQL query results from the SQL section of the Net.Data macro file in the output table, `MyTable`. This table is specified in the DEFINE section:

```
MyTable = %TABLE(all)
```

The call to the applet in the macro is specified in the HTML section:

```
@DTWA_MyGraph(MyTitle, DTW_COLUMN(ages) MyTable)
```

Based on the parameters in the function call, Net.Data generates the complete applet tag containing the information about the result table, such as the number of columns, the number of rows returned, and the result rows. Net.Data generates one parameter tag for each cell in the result table, as shown in the following example:

```
param name = 'DTW_MyTable_ages_VALUE_1' value = "35">
```

The parameter name, *DTW_MyTable_ages_VALUE_1*, specifies the table cell (row 1, column `ages`) in the table, `MyTable`, which has a value of 4. The keyword, DTW_COLUMN, in the function call to the applet, specifies that you are interested only in the column ages of the resulting table, MyTable, shown here:

```
@DTWA_MyGraph( MyTitle, DTW_COLUMN(ages) MyTable )
```

The following output shows the complete applet tag that Net.Data generates for the example:

```
<applet code = 'MyGraph.class'
codebase = '/netdata-java/' width = '400' height = '200' >
<param name = 'MyTitle' value = "This is my Title" >
<param name = 'DTW_NUMBER_OF_TABLES' value = "1" >
<param name = 'DTW_TABLE_1_NAME' value = "MyTable" >
<param name = 'DTW_MyTable_NUMBER_OF_ROWS' value = "5" >
<param name = 'DTW_MyTable_NUMBER_OF_COLUMNS' value = "1" >
<param name = 'DTW_MyTable_COLUMN_NAME_1' value = "ages" >
<param name = 'DTW_MyTable_ages_VALUE_1' value = "35">
<param name = 'DTW_MyTable_ages_VALUE_2' value = "32">
<param name = 'DTW_MyTable_ages_VALUE_3' value = "31" >
<param name = 'DTW_MyTable_ages_VALUE_4' value = "28" >
<param name = 'DTW_MyTable_ages_VALUE_5' value = "40" >
<P>Sorry, your browser is not Java-enabled.<BR>
</applet>
```

Using the Net.Data Java Applet Interface

Net.Data provides a set of interfaces in a class called DTW_Applet.class, which you can use with your Java applets to help process the PARAM tags that are generated for table variables. You can create an applet that extends this interface to call the routines from your applet.

Net.Data provides these interfaces:

- **int GetNumberOfTables()** returns the number of tables found in the applet tag.
- **String [] GetTableNames()** returns a list of the table names found in the applet tag.
- **int GetNumberOfColumns(String table_name)** returns the number of columns in the table table_name.
- **int GetNumberOfRows(String table_name)** returns the number of rows in the table table_name.
- **String[] GetColumnNames(String table_name)** returns the names of the columns in the table table_name.
- **String[][] GetTable(String table_name)** returns a two-dimensional array of strings containing the values of the table's rows and columns.

To access the interfaces, use the EXTENDS keyword in your applet code to subclass your applet from the DTW_APPLET class, as shown in the following example:

```
import java.io.*;
import java.applet.Applet;

public class myDriver extends DTW_Applet
    {
    public void init()
        {
        super.init();

        if (GetNumberOfTables() > 0)
            {
            String [] tables = GetTableNames();
            printTables(tables);
            }
        }

    private void printTables(String[] tables)
        {
        String table_name;

        for (int i = 0; i < tables.length; i++)
        {
        table_name = tables[i];
        printTable(table_name);
        }
    }

private void printTable(String table_name)
    {
    int nrows = GetNumberOfRows(table_name);
    int ncols = GetNumberOfColumns(table_name);

    System.out.println("Table: " + table_name + " has " + ncols + " columns and
                         " + nrows + " rows.");

    String [] col_names = GetColumnNames(table_name);

    System.out.println("------------------------------------------------------");

    for (int i = 0; i < ncols; i++)
        System.out.print("   " + col_names[i] + "   ");
    System.out.println("\n----------------------------------------------------");

    String [][] mytable = GetTable(table_name);

    for (int j = 0; j < nrows; j++)
        {
        for (int i = 0; i < ncols; i++)
            System.out.print("   " + mytable[i][j] + "   ");

        System.out.println("\n");
        }
    }
}
```

Java Application Language Environment

Net.Data supports your existing Java applications with the Java language environment. With support for Java applets and Java methods (or applications), you can access DB2 through the Java Database Connectivity (JDBC**) API.

Details about JDBC are available from these Web sites:

- IBM Software has JDK 1.1 or higher, which is required to use JDBC with Net.Data:

  `http://www.software.ibm.com/data/db2/jdbc/`

- JavaSoft has additional JDBC drivers, JDBC API documentation, and the latest updates of JDBC:

  `http://splash.javasoft.com/jdbc/`

The Java language environment provides a Remote Procedure Call (RPC)-like interface. You can issue Java function calls from your Net.Data macro file with Net.Data strings as parameters and your invoked Java function can return a string. You must use the Net.Data Live Connection when you use the Java language environment (see the performance chapter of *Net.Data Administration and Programming Guide* for more information about Live Connection). In order to use the Java language environment you must complete the following steps. These steps are described in detail in subsequent sections.

1. Write your Java functions.
2. Create a Net.Data cliette for all your Java functions (Net.Data cliettes launch the Java Virtual Machine where your Java function runs.

3. Define a cliette statement in the Live Connection configuration file.
4. Start Connection Manager.
5. Run the Net.Data macro file that invokes the Java language environment.

Each time you introduce new Java functions, you must recreate the Java cliette.

Java Language Environment File Structure

Net.Data creates several directories during the Net.Data installation. These directories include the files you need to create your Java functions, define the cliette, and run the macro with the Java language environment:

- A sample Java function called `UserFunctions.java`.
- A sample file called `makeClas`. When run, this file creates a Net.Data cliette class for your Java function.
- A sample file called `launchjv` used by the Net.Data cliette to launch the Java Virtual Machine and run your Java function.

Table 2 describes the directory and file names for the files on your operating system.

Table 2. The Files Used for Creating Java Functions

Operating System	File name	Directory
OS/2	UserFunctions.java	javaapps
	launchjv.com	connect
Windows NT	UserFunctions.java	javaclas
	makeClas.bat	javaclas
	launchjv.bat	connect
UNIX	UserFunctions.java	javaapps
	launchjv	javaapps

Creating the Java Function

Modify the Java function sample file `UserFunctions.java`, or create a new file modeled on the following example file, called `myfile.java`:

```
====================myfile.java====================
import mypackage.*                                  <=contain your functions
 public String myfctcall(...parameters from macro file...)
{
 return ( mypackage.mymethod(...parameters...));    <=high-level call to your functions
 }

public String lowlevelcall(...parameters...)
{
 string result;
  .......code using many functions of your package...
  return(result)
}
```

Defining the Java Language Environment Cliette

Modify the sample file, `makeClas.bat`, or create a new .bat file to generate a Net.Data cliette class, called `dtw_samp.class`, for all your Java functions. The following example shows how the batch file, `CreateServer`, processes three Java functions:

```
rem Batch file to create dtw_samp for Net.Data
java CreateServer dtw_samp.java UserFunctions.java myfile.java
javac dtw_samp.java
```

The batch file processes the following files, along with the Net.Data-supplied stub file called `Stub.java` to create `dtw_samp.class`.

- `dtw_samp.java`
- `UserFunctions.java`
- `myfile.java`

Writing a JDBC application or applet is very similar to writing a C application using DB2 CLI or ODBC to access a database. The primary difference between applications and applets is that an application might require special software to communicate with DB2, for example, DB2 Client Application Enabler. The applet depends on a Java-enabled Web browser, and does not require any DB2 code installed on the client.

Your system requires some configuration before using JDBC. These considerations are discussed at the DB2 JDBC Application and Applet Support Web site:

`http://www.software.ibm.com/data/db2/jdbc/db2java.html`

Configuring Net.Data for the Java Language Environment

To use the Java language environment, you must configure Net.Data. Use the following steps to complete these configuration steps:

1. Create a batch file to launch the Java application because Net.Data cannot directly start a Java application. Net.Data uses this file to launch the Java Virtual Machine, which runs your Java function. The batch file must include the java-classpath statement to ensure the required Java packages (the standard and application-specific packages) can be found. For example, the batch file, `launchjv.bat`, contains the following java-classpath:

   ```
   java -classpath %CLASSPATH%;C:\DB2WWW\Javaclas dtw_samp %1 %2 %3 %4 %5 %6
   ```

2. Define a cliette to work with the Java language environment in the Live Connection configuration file, `dtwcm.cnf`. Specify unique port numbers for the cliette and the related batch file name with the EXEC_NAME configuration variable. In the following example, the Java cliette name is defined as DTW_JAVAPPS and the EXEC_NAME configuration variable is set to the name of the batch file, `launchjv.bat`:

   ```
   CLIETTE DTW_JAVAPPS{
   MIN_PROCESS=1              <= Required: this value must be 1 because
                               the JAVAPPS cliette is multi-threaded.
   MAX_PROCESS=1              <= Required: this value must be 1 because
                                 the JAVAPPS cliette is multi-threaded.
   START_PRIVATE_PORT=5100    <= Must be a unique port number
   START_PUBLIC_PORT=5300     <= Must be a unique port number
   EXEC_NAME=launchjv.bat     <= The name of the batch file that includes the
                               classpath statements
   }
   ```

When you start the Net.Data Connection Manager, Net.Data starts the Java cliette specified in the configuration file. The cliette becomes available to process Java language environment requests from your Net.Data macro applications.

3. Update the DTW_JAVAPPS ENVIRONMENT path statement in the Net.Data initialization file, `db2www.ini`, by adding each cliette name to the statement. For example:

```
ENVIRONMENT DTW_JAVAPPS  ( OUT RETURN_CODE ) CLIETTE "DTW_JAVAPPS"
```

Creating and Running the Macro File

After you have created the Java function, defined the cliette class, and configured Net.Data, you can run the macro file containing references to the Java function.

1. Create a macro file that calls your Java functions. For example, the function call, *myfctcall* calls the sample function provided with Net.Data, using the cliette DTW_JAVAPPS.

```
%function (DTW_JAVAPPS) myfctcall( ....parameters from macro file ....)

%{ to call the sample provided with Net.Data %}
%function (DTW_JAVAPPS) reverse_line1(str);

%HTML_REPORT{
you should see the string "Hello World" in reverse.
@reverse_line("Hello World")
You should have the result of your function call.
@myfctcall( ... ....)
%}
```

2. Start the Connection Manager. See the performance chapter of *Net.Data Administration and Programming Guide* for more information about Connection Manager.
3. Launch the macro and invoke Net.Data using an HTML link, HTML form, or URL statement. For example, invoke the Net.Data macro file, `mymacro.d2w`, with the following URL statement:

```
http://myserver/cgi-bin/dt2www/mymacro.d2w/report
```

ODBC Language Environment

The Open Database Connectivity (ODBC) language environment executes SQL statements through an ODBC interface. ODBC is based on the X/Open SQL CAE specification, which lets a single application access many database management systems.

To use the ODBC language environment, you must have an ODBC driver and a driver manager. Your ODBC driver documentation describes how to install and configure your ODBC environment.

Sending SQL statements in an ODBC environment is similar to other Net.Data functions. The following example is a Net.Data macro that sends multiple SQL statements to the database that is your ODBC data source. Operating systems that use the DATABASE variable must specify the same database as the data source in the ODBC.INI file.

```
%DEFINE {
   DATABASE="qesq1"
   SHOWSQL="YES"
   table="int_null"
   LOGIN="netdata1"
   PASSWORD="ibmdb2"
%}

%function(dtw_odbc) sql1() {
create table int_null (int1 int, int2 int)
%}

%function(dtw_odbc) sql2() {
insert into $(table) (int1) values (111)
%}

%function(DTW_odbc) sql3() {
insert into $(table) (int2) values (222)
%}

%function(dtw_odbc) sql4() {
select * from $(table)
%}

%function(dtw_odbc) sql5() {
drop table $(table)
%}

%HTML(REPORT) {
@sql1()
@sql2()
@sql3()
@sql4()
%}
```

Oracle Language Environment

The Oracle language environment provides native access to your Oracle data. You can access Oracle tables from Net.Data running in CGI, FastCGI, NSAPI, ISAPI, or GWAPI mode. This language environment supports Oracle 7.2, 7.3, and 8.0.

Restrictions:

- Stored procedures are not supported through this language environment.
- The DATABASE variable is not used to access Oracle databases.
- The LOGIN variable must contain the Oracle database instance name. For example, *ora73* is the defined instance name in the following LOGIN variable:

  ```
  LOGON=admin@ora73
  ```

To access Oracle from Net.Data

1. Verify that the ENVIRONMENT statement in the Net.Data initialization file is correct for the Oracle language environment. See the configuration chapter in *Net.Data Administration and Programming Guide* for steps and examples.
2. Ensure the appropriate components of Oracle are installed and working as follows:
   a. Install SQL*Net on the machine where Net.Data is installed, if it is not already installed. For more information, see the following URL:

      `http://www.oracle.com/products/networking/html/stnd_sqlnet.html`

   b. Verify that the Oracle *tnsping* function can be used with the same security authorization that your Web server uses. To verify, log on with your Web server's user ID and type:

      ```
      tnsping oracle-instance-name
      ```

Where *oracle-instance-name* is the name of the Oracle system that your Net.Data macros access.

You might not be able to verify the *tnsping* function on Windows NT if your Web server runs under system authority. If so, skip this step.

c. Verify that the Oracle tables can be accessed with the same security authorization that your Web server uses. To verify, enter an SQL SELECT statement, using the SQL*Plus line command tool, to access an Oracle table with an SQL SELECT statement with the authority of your Web server. For example:

```
SELECT * FROM tablename
```

You might not be able to verify table access on Windows NT if your Web server runs under system authority. If so, skip this step.

Troubleshooting: Do not proceed if the above steps fail. If any of the steps fail, check your Oracle configuration.

3. Ensure that the Oracle environment variables are set correctly in your Web server process.
   - For AIX, put the following lines in the `/etc/environment` file:

```
ORACLE_SID=oracle-instance-name
ORACLE_HOME=oracle-runtime-library-directory
```

   - For Windows NT, use the System Properties Control panel to add the following environment variables:

```
ORACLE_SID=oracle-instance-name
ORACLE_HOME=oracle-runtime-library-directory
```

Hint: You might require additional lines for other Oracle environment variables, depending on the Oracle facilities you plan to use, such as national language support and two phase commit. Consult the Oracle administration documentation for more information on these environment variables.

4. Test the connection to Oracle from Net.Data. In your Net.Data macro file, specify the appropriate values in the LOGIN and PASSWORD variables. *Do not* define the DATABASE variable when accessing Oracle databases. The following is an example of connect statement in a macro file:

```
%DEFINE LOGIN=user_ID@remote-oracle-instance-name
%DEFINE PASSWORD=password
```

Local Oracle instances:

If you access the local Oracle instance only, do not specify the remote-oracle-instance name as part of the login user ID, as in the following example:

```
%DEFINE LOGIN=user_ID
%DEFINE PASSWORD=password
```

Live Connection:

If you use Live Connection, then you can specify the LOGIN and PASSWORD in the Live Connection configuration file, although it is not recommended for security purposes. For example:

```
LOGIN=user_ID
PASSWORD=password
```

Hint: Do not specify the DATABASE variable for Oracle.

5. Test your configuration by running a CGI shell script to ensure that the Oracle instance can be accessed from your Web server, as in the following example:

   ```
   #! /bin/sh
   echo "content-type; text/html
   echo
   echo "< html>< pre>"
   set
   echo "</pre>< p>< pre>"
   tnsping oracle-instance-name
   echo
   ```

   Alternatively, you can execute *tnsping* directly from a Net.Data macro, as in the following example:

   ```
   %DEFINE testora = %exec "tnsping oracle-instance-name"
   %HTML (report){
   < P>About to test Oracle access with tnsping.
   < hr>
   $(testora)
   < hr>
   < P>The Oracle test is complete.
   %}
   ```

   Troubleshooting:

   If the verification step fails, check that all the preceding steps were successful by verifying the following items:

   - Check your Oracle configuration.
   - Verify that the Oracle environment variable syntax is correct and that no variables are missing.
   - Check the Oracle connection, ensuring that you have entered the correct user ID and password.

   If the verification step still fails, contact IBM Service.

Example:

After you have completed the accessing verification steps, you can make calls to the Oracle language environment with functions in the macro file, as in the following example:

```
%FUNCTION(DTW_ORA) STL1() {
insert into $(tablename) (int1,int2) values (111,NULL)
%}
```

Perl Language Environment

The Perl language environment can interpret inline Perl scripts that you specify in a FUNCTION block of the Net.Data macro, or it can process external Perl scripts that are stored in separate files on the server. Calls to external Perl scripts are identified in a FUNCTION block by an EXEC statement, for example:

```
%EXEC{ perl-script-name [optional parameters] %}
```

The Perl language environment cannot directly pass or retrieve Net.Data variables, so they are made available to Perl scripts in this manner:

- Net.Data passes input parameters to the Perl script as environment variables. The Perl script can retrieve the parameters by reading the Perl associative array.

- The Perl script passes output parameters back to the language environment by writing to a named pipe whose name Net.Data passes in the environment variable, DTWPIPE. Use the DEFINE statement syntax to write data to the named pipe:

  ```
  name = value
  ```

  For multiple data items, separate each item with a new-line or blank character.

  If a variable name has the same name as an output parameter and uses the above syntax, the new value replaces the current value. If a variable name does not correspond to an output parameter, Net.Data ignores it.

The following example shows how Net.Data passes variables from a macro file.

```
%FUNCTION(DTW_PERL) today() RETURNS(result) {
  $date = 'date';
  chop $date;
  open(DTW, "> $ENV{DTWPIPE}") || die "Could not open: $!";
  print DTW "result = \"$date\"\n";
%}
%HTML(INPUT) {
  @today()
%}
```

If the Perl script is in an external file called `today.prl`, the same function can be written as in the next example:

```
%FUNCTION(DTW_PERL) today() RETURNS(result) {
  %EXEC { today.prl %}
%}
```

A Perl language environment program accesses the values of a table parameter by their Net.Data name. The column headings for table T are T_N_*i*, and the field values are T_V_*i_j*. The number of rows and columns in table T are T_ROWS and T_COLS.

REPORT and MESSAGE blocks are permitted as in any FUNCTION section. They are processed by Net.Data, not by the language environment. A Perl program can, however, write text to the standard output stream and directly manipulate the output HTML form.

Authorization Tip: Ensure that the Web server has access rights to any external executable files referenced by this language environment, including the correct version of the Perl interpreter. See the section on specifying Web server access rights to Net.Data files in the configuration chapter of *Net.Data Administration and Programming Guide* for more information.

REXX Language Environment

The REXX language environment can interpret inline REXX programs that are specified in a FUNCTION block of the Net.Data macro, or it can execute external REXX programs stored in separate files. Calls to external REXX programs are identified in a FUNCTION block by a statement, for example:

```
%EXEC{ REXX-program-file-name [optional parameters] %}
```

The REXX language environment uses the RexxStart() API to tell the REXX interpreter to execute the specified file, then passes the parameters following the file name to the program as if they were entered on the command line. To the REXX program, all of the parameters are received as ARG[1].

Authorization Tip: Ensure that the Web server has access rights to any external executable files referenced language environments. See the section on specifying Web server access rights to Net.Data files in the configuration chapter of *Net.Data Administration and Programming Guide* for more information.

Variable Substitution:

Variable substitution is performed only on the executable-statements section of the FUNCTION block. Parameters, however, are made accessible to the REXX program whether the program is defined internally in a FUNCTION block or externally in a separate file. The REXX language environment uses the REXX language processors' RexxVariablePool() function to share Net.Data variables with the REXX program. This allows the REXX program to directly manipulate the Net.Data variables identified in the parameter list.

A REXX program accesses the values of a table parameter as REXX stem variables. To a REXX program, the column headings for table T are T_N.*i* and the field values are T_V.*i.j*. The number of rows and columns in table T are T_ROWS and T_COLS.

Improving Performance for the AIX operating system:

If you have many calls to the REXX language environment on your AIX system, consider setting the RXQUEUE_OWNER_PID environment variable to 0. Macros that make many calls to the REXX language environment can easily spawn many processes, swamping system resources.

You can set the environment variable in one of three ways:

- In the macro file by using the DTW_SETENV built-in function:
  ```
  @DTW_rSETENV("RXQUEUE_OWNER_PID", "0")
  ```
- In the AIX system environment file by inserting the following statement:
  ```
  /etc/environment: RXQUEUE_OWNER_PID = 0
  ```

  This method affects the behavior of REXX for the whole machine.
- In the HTTP Web server environment file; for example, for the Domino Go Webserver, insert the following statement:
  ```
  InheritEnv RXQUEUE_OWNER_PID = 0
  ```

  This method affects the behavior of REXX for the Web server.

SQL Language Environment

The SQL language environment is used to execute SQL statements through DB2. The results of the SQL statement can be returned in the Net.Data default table or in a table specified by you.

Net.Data supports any SQL statement you authorize. You can connect to one database for each HTML section when invoking Net.Data as a CGI application and must specify the database name with the DATABASE variable (except with OS/390). If your DB2 database is on the same machine as the Web server, no additional setup is required. Otherwise, depending on the operating system you use, you can access remote databases by using Client Application Enabler (CAE) or use Database Connection Services (DDCS) to get all the transaction support DB2 supports. You might also be able to use DataJoiner to access other databases. Using DataJoiner lets you use two-phase commit with databases that support it.

Sybase Language Environment

The Sybase language environment provides native access to your Sybase data. You can access Sybase tables from Net.Data running in CGI, FastCGI, NSAPI, ISAPI, or GWAPI mode.

Restrictions:

- The Sybase language environment does not support large objects, such as images and audio. Stored procedures are supported only for procedures without a SELECT statement.
- The Sybase language environment requires Live Connection to use FastCGI.

To access Sybase from Net.Data

1. Verify that the ENVIRONMENT statement in the Net.Data initialization file is correct for the Sybase language environment. See the configuration chapter in *Net.Data Administration and Programming Guide* for steps and examples.
2. Ensure the appropriate components of Sybase are installed and working as follows:
   a. Install Sybase's Open Client on the machine where Net.Data is installed, if it is not already installed. For more information, see the Sybase Open Client documentation for more information.
   b. Verify that the Sybase *ping* function can be used with the same security authorization that your Web server uses. To verify, log on with your Web server's user ID and type:
      ```
      ping sybase-instance-name
      ```
      Where *sybase-instance-name* is the name of the Sybase system that your Net.Data macros access.

      You might not be able to verify the *ping* function on Windows NT if your Web server runs under system authority. If so, skip this step.
   c. Verify that the Sybase tables can be accessed with the same security authorization that your Web server uses. To verify, enter an SQL SELECT statement, using the ISQL line command tool, to access an Sybase table with the authority of your Web server. For example:
      ```
      SELECT * FROM tablename
      ```
      You might not be able to verify table access on Windows NT if your Web server runs under system authority. If so, skip this step.

   Troubleshooting: Do not proceed if the above steps fail. If any of the steps fail, check your Sybase configuration.
3. Ensure that the Sybase environment variables are set correctly in your Web server process.
   - For AIX, put the following lines in the `/etc/environment` file:
     ```
     DSQUERY=sybase-instance-name
     SYBASE=sybase-runtime-library-directory
     ```

- For Windows NT, use the System Properties Control panel to add the following environment variables:

  ```
  DSQUERY=sybase-instance-name
  SYBASE=sybase-runtime-library-directory
  ```

Hint: You might require additional lines for other Sybase environment variables, depending on the Sybase facilities you plan to use, such as national language support and two-phase commit. Consult the Sybase administration documentation for more information on these environment variables.

4. Test the connection to Sybase from Net.Data. In your Net.Data macro file, specify the appropriate values in the LOGIN, PASSWORD, and DATABASE variables. The following is an example of connect statement in a macro file:

   ```
   %DEFINE DATABASE=database-name
   %DEFINE LOGIN=user_ID@remote-sybase-instance-name
   %DEFINE PASSWORD=password
   ```

   Live Connection: If you use Live Connection, then you can specify the LOGIN and PASSWORD in the Live Connection configuration file, although it is not recommend for security purposes. For example:

   ```
   DATABASE=database-name
   LOGIN=user_ID
   PASSWORD=password
   ```

5. Test your configuration by running a CGI shell script to ensure that the Sybase instance can be accessed from your Web server, as in the following example:

   ```
   #! /bin/sh
   echo "content-type; text/html
   echo
   echo "< html>< pre>"
   set
   echo "</pre>< p>< pre>"
   isql -u user_ID -p password << EOFF
   SELECT * FROM tablename
   EOFF
   echo
   ```

   Troubleshooting:

   If the verification step fails, check that all the preceding steps were successful by verifying the following items:

   - Check your Sybase configuration.
   - Verify that the Sybase environment variable syntax is correct and that no variables are missing.
   - Check the Sybase connection, ensuring that you have entered the correct user ID and password.

   If the verification step still fails, contact IBM Service.

Example:

Once you have completed the accessing verification steps, you can make calls to the Sybase language environment with functions in the macro file, as in the following example:

```
%function(DTW_SYB) STL1() {
insert into $(tablename) (int1,int2) values (111,NULL)
%}
```

System Language Environment

The System language environment is a Net.Data-defined environment that supports calls to external programs identified in an EXEC statement in the FUNCTION block.

The System language environment interprets the EXEC statement by passing the program name and parameters to the operating system for processing using the C language system() function call. This method does not allow the external program to directly pass variables to and from Net.Data, as the REXX language environment does, so Net.Data processes the variables in the following method:

- Net.Data passes input parameters to the external program as environment variables and the external program retrieves them:
  - A UNIX CSHELL script refers to environment variables by preceding the environment variable name with a dollar sign ($), such as `$x`.
  - A Perl language script refers to them by referring to the associative array ENV, such as `%ENV{'x'}`.
  - A DOS batch file refers to the variable name enclosed in percent signs (%), such as `%x%`.
- Most operating systems pass output parameters back to the language environment by writing to a named pipe whose name Net.Data passes in the environment variable, DTWPIPE. (Net.Data for OS/400 passes parameters back to the language environment using environment variables.) Use the DEFINE statement syntax to write the data to the named pipe:

  ```
  name = value
  ```

  For multiple data items, separate each item by a new-line or blank character.

  If a variable name matches an output parameter, the new value replaces the current value. Net.Data ignores variable names that do not match any output parameters.

A system language environment program accesses the values of a table parameter by the Net.Data name. The column headings for table T are T_N_*i*, and the field values are T_V_*i*_*j*. The number of rows and columns in table T are T_ROWS and T_COLS.

Authorization Tip: Ensure that the Web server has access rights to any external executable files invoked from the System language environment. See the section on specifying Web server access rights to Net.Data files in the configuration chapter of *Net.Data Administration and Programming Guide* for more information.

Web Registry Language Environment

The Net.Data Web registry provides persistent storage for application-related data. A Web registry can be used to store configuration information and other data that can be accessed dynamically at run time by Web-based applications. You can access Web registries only through Net.Data macros using Net.Data and the Web registry built-in support and from CGI programs written for this purpose. The Web registry available on a subset of operating systems. See the Net.Data operating system reference appendix in *Net.Data Reference*

Standard Web page development requires that URLs be placed directly in the HTML source for the page. This makes changing links difficult. The static nature also limits the type of links that can be easily placed on a Web page. Using a Web registry to store application-related data, for example URLs, can help in the creation of HTML pages with dynamically set links.

Information can be stored and maintained in a registry by application developers and Web administrators who have write access to the registry. Applications retrieve the information from their associated registries at run time. This facilitates the design of flexible applications and also allows movement of applications and servers. You can use Net.Data macros to create HTML pages using dynamically set links.

Information is stored in a Web registry in the form of registry entries. Each registry entry consists of a pair of character strings: a RegistryVariable string and a corresponding RegistryData string. Any information that can be represented by a pair of strings can be stored as a registry entry. Net.Data uses the variable string as a search key to locate and retrieve specific entries from a registry.

You can see sample contents of a Web registry in Table 3.

Table 3. Sample Web Registry

CompanyName	**WorldConnect**
Server	ftp.einet.net
JohnDoe/foreground	Green
CompanyURL/IBM Corp.	http://www.ibm.com
CompanyURL/Sun Microsystems Corp.	http://www.sun.com
CompanyURL/Digital Equipment Corp.	http://www.dec.com
JaneDoe/Home_page	http://jane.info.net

Here are some reasons to consider using a Web registry:

- You can use a Web registry to store aliases for servers and URLs, facilitating the relocation of applications and servers.
- Application developers can ship their Web-based applications with data, such as URLs, predefined in the registry. The end user can modify the registry data to change the behavior of the application.
- A Web registry can be used to perform URL searches based on product name, national language, manufacturer, and so on.

Indexed entries in the Web Registry are entries whose RegistryVariable strings have an additional Index string appended to them, using the following syntax:

```
RegistryVariable/Index
```

The user provides the value of the index string in a separate parameter to a built-in function designed to work with indexed entries. Multiple indexed registry entries can have the same RegistryVariable string value, but they can maintain their uniqueness by having different Index string values.

Table 4. Sample Indexed Web Registry

Smith/Company_URL	http://www.ibmlink.ibm.com
Smith/Home_page	http://www.advantis.com

Even though the above two indexed entries have the same RegistryVariable string value `Smith`, the index string is different in each case. They are treated as two distinct entries by the Web registry functions.

Configuring Net.Data Language Environments

Before you can use the language environments with Net.Data, you must configure Net.Data initialization file, and if you are using Live Connection, the Live Connection configuration file.

The following is an overview of these tasks. See the configuring chapter in *Net.Data Administration and Programming Guide* for detailed information about how to configure the Net.Data language environments. Additionally, check the previous language environment sections for special configuration instructions.

- Verify or update the ENVIRONMENT statement in the Net.Data initialization file, `db2www.ini`. These statements are described in the “Configuring Net.Data” chapter in *Net.Data Administration and Programming Guide*.
- Define cliettes for database or the Java applications language environments in the Live Connection file, `dtwcm.cnf`. Defining cliettes is described in the configuring chapter in *Net.Data Administration and Programming Guide*.

Part 2. Non-IBM Language Enviroments

In addition to supplying language environments, Net.Data enables you to add language and database environments of your own. Net.Data accesses your language environments as dynamic link libraries or shared libraries, separate from the Net.Data executable file. Each language environment must support a set of interfaces defined by Net.Data. The following chapters discuss how to create a new language environment and describe the language environment programming interface and environments.

"Chapter 3. Creating a New Language Environment" on page 31

"Designing the Language Environment Statement" on page 39

"Chapter 4. The Language Environment Programming Interface Utility Functions" on page 43

Chapter 3. Creating a New Language Environment

Net.Data uses language environments as pluggable programming language and database interfaces, accessed as DLL files or shared libraries. Net.Data provides a set of language environments, but when these do not meet your applications needs, you can create new language environments. Before you decide to create a new language environment, determine if the IBM-supplied language environments that are shipped with Net.Data satisfy your needs.

When you decide to create a new language environment, then you must complete the following steps:

- Determine what interfaces and functions you must provide for the language environment. The dtw_execute() interface must be provided, and all provided interfaces must match exactly the prototypes that are defined in the dtwle.h C language header.
- Create a makefile or JCL to build the DLL or shared library.
- Build a DLL or shared library that implements the set of language environment interface routines you want to provide. See the C or C++ documentation for your operating system to learn how to create makefiles and build DLLs or shared libraries.
- Make all interfaces externally available from the DLL or shared library so Net.Data can call them.
- Determine your ENVIRONMENT configuration statement, then add it to the Net.Data initialization file. See “Designing the Language Environment Statement” on page 39 for details.
- Add functions to the Net.Data macro file that uses the new language environment.

This chapter describes how to design the language environment.

- “Designing a DLL or Shared Library”
- “Language Environment Communication Structures” on page 33
- “Language Environment Interface Functions” on page 36
- “Designing the Language Environment Statement” on page 39

Designing a DLL or Shared Library

When you build a language environment, you update the template provided in “Appendix A. Language Environment Template” on page 71 to include the environment interface functions and the communication structures used by Net.Data to communicate with your language environment and to pass parameters to and from the language environment.

The following sections describe concepts and design issues for the functions and structures. The utilities provided in the language environment interface are described in “Chapter 4. The Language Environment Programming Interface Utility Functions” on page 43.

- “Which Language Environment Interfaces Should I Provide?” on page 32
- “Processing Input Parameters” on page 32

- “Processing User Requests” on page 33
- “Processing Output Parameters” on page 33
- “Communicating Error Conditions” on page 33

Which Language Environment Interfaces Should I Provide?

When you write a language environment, you must determine which interfaces to provide. Your choices depend on what you intend the language environment to do. For example, if the language environment will be accessing database data, you’ll make different choices than if it is for a scripting language. The following section describes the Net.Data language environment interfaces.

dtw_execute()
You must provide the dtw_execute() interface to pass input parameters from the macro file; it is the only required interface for every language environment. Net.Data passes all input parameters to dtw_execute() through the language environment communication structure, dtw_lei .

dtw_initialize()
Provide the dtw_initialize() interface to allocate or initialize data. Net.Data calls this interface only once for each macro invocation, before the first function call to your language environment. If there are no function calls to your language environment, Net.Data does not call the dtw_initialize() interface.

dtw_cleanup()

Provide the dtw_cleanup() interface when you provide a dtw_initialize() interface, and you want to allow for error handling when the macro terminates abnormally. Net.Data calls this interface only once for each macro invocation.

dtw_getNextRow()
Provide the dtw_getNextRow() interface as part of a database language environment or a language environment that can process data a row at a time. This interface is called if Net.Data is running on the OS/400 or OS/390 operating systems, only.

Processing Input Parameters

The Net.Data language environments use the dtw_execute() interface to receive and process parameters. The dtw_execute() interface works with the dtw_lei structure, which is used by Net.Data to communicate with the language environment. Use the following recommendations for input parameter processing, when writing your language environment.

- Specify any implicit parameters in the initialization file. Net.Data passes the parameters specified here on all function calls to the language environment after it passes the parameters specified by the macro writer on the FUNCTION block being executed.
- Receive input parameters to the dtw_execute() interface as part of the dtw_lei structure. The macro writer determines the order that Net.Data passes the parameters when specifying them in the FUNCTION block definition of the Net.Data macro.

The processInputParms() routine in the program template, in “Appendix A. Language Environment Template” on page 71 shows one method of processing input parameters.

Processing User Requests

How a language environment processes a user request depends on how the language environment receives the request. Net.Data provides several different ways for you to communicate a request to your language environment:

- Through the function name specified on a FUNCTION block. On every function call, Net.Data passes the function name to the language environment in the function_name field of the dtw_lei structure.
- Through the FUNCTION block parameter list. You can specify that a parameter in the parameter list can indicate a user request. On every function call, Net.Data passes parameters to the language environment in the parm_data_array field of the dtw_lei structure.
- Through the executable-statements section of a FUNCTION block. On every function call, Net.Data passes any executable statements specified in the FUNCTION block to the language environment in the exec_statement field of the dtw_lei structure.

Processing Output Parameters

The method you use to process output parameters depends entirely on your language environment and how it processes user requests. However, once the language environment has the data it needs to return to the Net.Data macro, you can design the language environment to modify the values of parameters passed in the parm_data_array field of the dtw_lei structure. The processOutputParms() routine in the program template, in “Appendix A. Language Environment Template” on page 71, shows one possible way of processing output parameters, as well as examples of how to set both string and table parameter values.

Communicating Error Conditions

The success or failure of a function call can be communicated through the implicit Net.Data macro variable, RETURN_CODE. This variable is set by Net.Data after returning from a call to the dtw_execute() interface. Its value is set to the return value of the dtw_execute() call itself. This value is then used by Net.Data to process the Net.Data macro MESSAGE block, if one was specified for this function call.

If you do not specify a MESSAGE block, or do not have an entry in a specified MESSAGE block to handle the return code from dtw_execute(), Net.Data displays the contents of the default_error_message field of the dtw_lei structure. This field can be set by the language environment at any time in the dtw_execute() routine. The setErrorMessage() routine in the program template, in “Appendix A. Language Environment Template” on page 71, shows an example of how to set the default_error_message field.

Language Environment Communication Structures

Net.Data uses two structures to communicate with your language environment. Your language environment must work with these structures and set and pass information within the structures.

- dtw_lei
- dtw_parm_data

Net.Data passes a language environment interface structure (for example, dtw_lei) to the language environment function that it calls. The structure contains, among other things, a parameter data array that contains a list of parameters to be passed to the language environment function. The language environment function called by Net.Data processes the request, updates the parameters in the parameter data array (if applicable), and returns to Net.Data.

Net.Data then goes through the parameter data array, updates its copies of the parameters to reflect the new values set by the language environment function, and continues the processing of the Net.Data macro.

The dtw_lei Structure

The interface function of each language environment receives a pointer to the dtw_lei structure. The dtw_lei structure has the following format:

```
typedef struct dtw_lei {              /* Lang. Env. Interface       */
    char *function_name;              /* Function block name        */
    int   flags;                      /* Lang. Env. Interface flags  */

    char *exec_statement;             /* Lang. Env. statement(s)     */

    dtw_parm_data_t *parm_data_array; /* Parameter array             */
    char *default_error_message;      /* Default message             */
    void *le_opaque_data;             /* Lang. Env. specific data    */

    void *row;                        /* For row-at-a-time processing*/

    char  reserved[64];               /* Reserved                    */
} dtw_lei_t;
```

Fields in the dtw_lei structure:

function_name
The function_name field contains a pointer to a string containing the name of the function block. This can be useful to specify the FUNCTION block name in error messages displayed by the language environment.

flags The flags field is used by Net.Data to communicate with the language environment. Specify the flags field pointer by performing an OR operation using the following constants:

- Net.Data sets DTW_STMT_EXEC to tell the dtw_execute() interface function that the exec_statement field contains the file name and parameters from an EXEC statement.
- DTW_END_ABNORMAL is set by Net.Data to tell the dtw_cleanup() interface function that an abnormal or unexpected condition has occurred and that the language environment should perform any cleanup necessary (that is, free held resources) before Net.Data ends.
- DTW_LE_FATAL_ERROR is set by a language environment interface function to tell Net.Data that a fatal error has occurred in the language environment. If this flag is set, Net.Data stops processing the Net.Data macro, calls all active language environment’s dtw_cleanup() interface function with flags set to DTW_END_ABNORMAL, prints default message, and exits. The flag is checked only if a non-zero return value is returned on a language environment call.
- DTW_LE_MSG_KEEP is set by a language environment interface function to tell Net.Data that the storage pointed to by default_error_message should not be freed. If this constant is not set, Net.Data attempts to free the storage.

- DTW_LE_CONTINUE is set by the dtw_execute() interface function to tell Net.Data to call the dtw_getNextRow() interface function. Net.Data calls dtw_getNextRow() only if the flag is set and the return value from the call to the dtw_execute() interface function is zero.

exec_statement
The exec_statement field contains one of the following pointers:

- To a string containing the executable statements (after variable substitution) from the FUNCTION block
- To the file name and parameters from an EXEC statement

parm_data_array
The parm_data_array field contains a pointer to an array of dtw_parm_data structures. The array ends with a parm_data structure containing zeros. The dtw_parm_data structure is used by Net.Data to pass variables and the associated value to a language environment and to retrieve any changes to the variable value that may be made by the language environment. See "The dtw_parm_data Structure" for a description of the structure.

default_error_message
The default_error_message field is set by the language environment to a character string that describes an error condition. If the return value from a call to a language environment interface function is non-zero and the return value does not match the value of a message in a MESSAGE block, the default message is displayed. Otherwise, Net.Data displays the message selected from the MESSAGE block.

le_opaque_data
The le_opaque_data field is set by any of the interface functions in the language environment to pass parameters from one interface function to another. Net.Data saves the pointer and passes it to another interface function that Net.Data calls. After processing the Net.Data macro, and before returning to the caller of Net.Data, Net.Data defines the pointer to NULL. Because the field is thread-specific, language environments can store data that is thread specific. Use this field only if you have a dtw_cleanup() interface function, so that the function can free the storage associated with the le_opaque_data field.

row The row field is set by Net.Data to a row object prior to calling a language environment's dtw_getNextRow() interface function. The dtw_getNextRow() function inserts a row of table data in the object using the Net.Data row utility interface functions. Net.Data then processes the row and calls dtw_getNextRow() until there are no more rows to process.

The reserved field is reserved for IBM use.

The dtw_parm_data Structure

Net.Data uses the dtw_parm_data structure to pass parameters to a language environment. Parameters are obtained from three sources:

- Explicit parameters that are specified on the FUNCTION block definition
- Parameters that are specified on the ENVIRONMENT configuration statement in the Net.Data initialization file
- The return variable that is specified on the RETURNS keyword on a FUNCTION block definition

Net.Data passes explicit parameters first, followed by parameters specified in the ENVIRONMENT statement, and then the return variable.

The dtw_parm_data structure has the following format:

```
typedef struct dtw_parm_data {        /* Parameter data             */
    int   parm_descriptor;            /* Parameter descriptor        */
    char *parm_name;                  /* Parameter name              */
    char *parm_value;                 /* Parameter value             */
    void *res1;                       /* Reserved                    */
    void *res2;                       /* Reserved                    */
} dtw_parm_data_t;
```

Fields in the dtw_parm_data structure:

parm_descriptor

The parm_descriptor field describes the type and use of the parameter being passed to the language environment. Net.Data sets the field by performing an OR operation using the following constants:

- DTW_IN indicates that a parameter is an input-only parameter.
- DTW_OUT indicates that a parameter is an output-only parameter.
- DTW_INOUT indicates that a parameter is an input and output parameter.
- DTW_STRING indicates that parameter value is a pointer to a string.
- DTW_TABLE indicates that the parameter value is a pointer to a table.

  Net.Data always sets the parm_descriptor field to DTW_IN, DTW_OUT, or DTW_INOUT and uses a logical OR with DTW_STRING and DTW_TABLE.

parm_name

The parm_name field is a pointer to a string that contains the name of the parameter. Net.Data sets this pointer NULL if the parameter is a literal string.

parm_value

The parm_value field is a pointer to an object that contains the value of the parameter. This pointer is set to NULL by Net.Data if the parameter is a variable that is not already defined.

The res1 and res2 fields are reserved fields.

Both parm_name and parm_value point to an object allocated from the Net.Data run-time *heap*, the area of memory used for dynamic memory allocation by Net.Data. If parm_name or parm_value is replaced with another string, the original string must be freed and replaced with a pointer to a character string allocated from the Net.Data heap. Use the dtw_malloc() and dtw_free() utility functions to free the original string.

Language Environment Interface Functions

Net.Data uses four interface functions with a language environment: you provide one or more of these functions. Three of these functions are optional, but every language environment must have a dtw_execute() interface function. If a Net.Data macro references a language environment that does not have a dtw_execute() interface function, Net.Data returns an error message and stops processing the Net.Data macro.

To call a language environment, reference it on the FUNCTION block of the Net.Data macro. The language environment interface functions must be called in the following order:

1. dtw_initialize()
2. dtw_execute()
3. dtw_getNextRow()
4. dtw_cleanup()

The dtw_execute() function is the only interface function that you must provide in the language environment.

When Net.Data encounters a call to a function that uses the language environment, it uses the following steps to call the language environment:

1. Net.Data calls dtw_initialize() if it has been defined for this language environment. The function performs any initialization tasks required by the language environment, such as connecting to databases, or allocating variables.
2. Net.Data calls dtw_execute() to process the macro file FUNCTION block containing statements that the language environment must process.
3. Net.Data calls dtw_getNextRow() if, on successful return, dtw_execute() indicated that dtw_getNextRow() should be called.
4. When the Net.Data macro processing is complete, Net.Data calls dtw_cleanup() to clean up the environment (for example, disconnecting from the database or freeing variables) if this function has been defined for the language environment, and then returns to the Web server.

The following sections describe the interface functions:

- “dtw_initialize()”
- “dtw_execute()” on page 38
- “dtw_getNextRow()” on page 38
- “dtw_cleanup()” on page 39

dtw_initialize()

The dtw_initialize() interface function performs any special initialization that the language environment requires, such as connecting to a database or allocating variables. This interface function is called once and is optional.

Net.Data calls a language environment’s dtw_initialize() interface function only once, the first time Net.Data calls a FUNCTION block referencing that language environment. Subsequent references to the language environment bypass the call to the dtw_initialize() interface function.

This interface function does not affect message block processing. A positive or zero return code means that processing continues; a negative return code means that processing does not continue. If the return code is non-zero and there is a default message defined in the default_error_message field, the default message is issued; if there is no default message, Net.Data issues an error message.

dtw_execute()

The dtw_execute() interface function processes macro file FUNCTION blocks that contain statements that must be processed by the language environment. For example, a FUNCTION block that refers to a database language environment contains SQL statements that language environment uses to query the database.

The dtw_execute() interface function is called whenever a Net.Data macro processes a FUNCTION block that refers to the language environment. When the dtw_execute() interface function completes, what happens next depends on whether the language environment is processing table data a row at a time. If so, the interface function sets DTW_LE_CONTINUE flag in the dtw_lei structure to tell Net.Data to call the dtw_getNextRow() interface function. See “dtw_getNextRow()” for more information about the dtw_getNextRow() interface function and its processing steps.

You can optimize performance by having the dtw_execute() interface function do all the processing necessary to produce the input for the report block processing. For example, the SQL language environment’s dtw_execute interface function generates the entire table to be processed during the report block phase.

dtw_getNextRow()

The dtw_getNextRow() interface function retrieves input for row-at-a-time processing of Net.Data tables. It is called each time the DTW_LE_CONTINUE flag is set, indicating that another row of data needs to be processed for the table. Use dtw_getNextRow() for database language environments.

Restriction: This interface function is only called if Net.Data is running on the OS/400 or OS/390 operating systems.

Net.Data calls dtw_getNextRow() when the following conditions are met:

- The call to the language environment’s dtw_execute() call completes successfully (return value of zero)
- The dtw_execute() interface function has set the DTW_LE_CONTINUE flag in the dtw_lei structure.

When the dtw_execute() function sets the DTW_LE_CONTINUE flag to on, Net.Data performs the following steps:

1. Processes the message block for the return value of the dtw_execute() interface function.
2. Calls language environment’s dtw_getNextRow() interface function and begins row-at-a-time processing.
3. Processes the report block.
4. Processes the message block for the return value of the dtw_getNextRow() interface function.
5. Determines whether dtw_getNextRow() has turned on the DTW_LE_CONTINUE flag:
   - If yes, processing continues with the dtw_getNextRow() interface function in step 2.
   - If no, row-at-a-time processing ends and Net.Data continues processing the Net.Data macro.

When dtw_getNextRow() is called, the row field in the dtw_lei structure is set to point to a row object. To manipulate the row object, use the Net.Data utility functions, dtw_row_SetCols() and dtw_row_SetV(). Net.Data assumes that after the first call to the dtw_getNextRow() interface function the row object contains the column headings for the table. Subsequent calls contain the actual table data.

The dtw_getNextRow() function continues to get called (unless message block processing indicates otherwise) as long as the DTW_LE_CONTINUE flag is set.

dtw_cleanup()

Use the dtw_cleanup() interface function to cleanup the language environment if you use dtw_initialize() to initialize the language environment. For example, disconnecting from a database or freeing variables. This interface function is optional.

While handling a Net.Data request, Net.Data calls a language environment's dtw_cleanup() interface function once when either Net.Data processing ends or an error stops Net.Data from processing the macro file.

Net.Data sets the flags field in the dtw_lei structure to DTW_END_ABNORMAL if the cleanup processing is abnormal. The following abnormal conditions provide examples of when to use dtw_cleanup():

- A language environment interface function indicates that a fatal error occurred by setting the DTW_LE_FATAL_ERROR bit in the flags field in the dtw_lei structure.
- Net.Data encounters an unrecoverable error.
- The Net.Data macro message block processing results in an exit.

If a language environment's interface function sets the le_opaque_data field with a parameter to be passed between interface functions, use the dtw_cleanup() to free the field when processing ends.

This interface function does not affect message block processing. If the return value is non-zero, a default message is issued; if no default message exists, the macro processor issues a warning message.

Designing the Language Environment Statement

Each language environment has an ENVIRONMENT statement in the initialization file, DB2WWW.INI, that contains information specific to that language environment. When you create a new language environment, you need to design an environment statement for the initialization file and document how users should add it to the initialization file.

The ENVIRONMENT statements specify information about the language environment that Net.Data requires to call and load the language environment DLL or shared library, such as the language environment name, the DLL or shared library name, and the list of parameters to be passed to the language environment for each function call.

Net.Data reads the configuration information when it is invoked, but does not load language environment DLLs or shared libraries until a FUNCTION block identifying that language environment is called from within the macro file. The DLL remains loaded until Net.Data ends.

The following sections provide information about syntax, parameter descriptions, and examples that you can use in your documentation.

ENVIRONMENT Statement Syntax

An ENVIRONMENT statement has the following format:

```
ENVIRONMENT(type) library-name  ([usage parameter, ...)
```

Each ENVIRONMENT statement must be on a single line.

The following are the parameters you must specify for each language environment:

- *type*

  The name that associates this language environment with a FUNCTION block definition in a Net.Data macro. You must also specify the language environment type on a FUNCTION block definition to tell Net.Data which language environment processes the function call. The name cannot begin with the prefix `DTW_`. This prefix is reserved for language environments shipped with Net.Data. See the ″Function Block″ section in *Net.Data Reference* for more information about the FUNCTION block.
- *library_name*

  The name of the object containing the language environment interfaces that are called by Net.Data. In Windows NT and OS/2, the DLL name is specified without the *.dll* extension. In AIX, the name of the shared object is specified with the *.o* extension, and in OS/400, the service program name is specified with the *.SRVPGM* extension. OS/390 has no extensions for DLL files. Look at the initialization file shipped with Net.Data for your operating system to see how to specify this name. Consider using a fully qualified path name to make sure Net.Data finds the DLL or shared library.
- *parameter_list*

  The list of parameters that are passed to the language environment on each function call, in addition to those parameters specified in the FUNCTION block definition. They are passed in the *parm_data_array* field of the *dtw_lei* structure following the parameters specified in the FUNCTION block definition. You must define these parameters as variables in your Net.Data macro before the function call is made. If a function modifies the value of these parameters, the parameters retain the modified value once the function finishes processing.

ENVIRONMENT Statement Examples

The following examples show ENVIRONMENT statements for language environments that Net.Data supplies. These examples illustrate how to specify parameters. The variables you include in the ENVIRONMENT statements are ones that you want to allow Net.Data macro writers to define or override in their macros. See the operating system-specific information in the appendixes in *Net.Data Reference* or in your Net.Data README file or Program Directory for additional examples.

The following examples show syntax for OS/2, AIX, and Windows NT.

```
ENVIRONMENT (DTW_SQL)      DTWSQL    ( IN DATABASE, LOGIN, PASSWORD,
TRANSACTION_SCOPE, SHOWSQL, ALIGN, START_ROW_NUM, DTW_SET_TOTAL_ROWS)
ENVIRONMENT (DTW_SYB)      DTWSYB    ( IN DATABASE, LOGIN, PASSWORD,
TRANSACTION_SCOPE, SHOWSQL, ALIGN, START_ROW_NUM, DTW_SET_TOTAL_ROWS)
ENVIRONMENT (DTW_ORA)      DTWORA    ( IN LOGIN, PASSWORD,
TRANSACTION_SCOPE, SHOWSQL, ALIGN, START_ROW_NUM, DTW_SET_TOTAL_ROWS)
```

```
ENVIRONMENT (DTW_ODBC)     DTWODBC   ( IN DATABASE, LOGIN, PASSWORD,
TRANSACTION_SCOPE, SHOWSQL, ALIGN, DTW_SET_TOTAL_ROWS)
ENVIRONMENT (DTW_APPLET)   DTWJAVA   ( )
ENVIRONMENT (DTW_JAVAPPS)   ( OUT RETURN_CODE ) CLIETTE "DTW_JAVAPPS"
ENVIRONMENT (DTW_PERL)     DTWPERL   ( OUT RETURN_CODE )
ENVIRONMENT (DTW_REXX)     DTWREXX   ( OUT RETURN_CODE )
ENVIRONMENT (DTW_SYSTEM)   DTWSYS    ( OUT RETURN_CODE )
ENVIRONMENT (HWS_LE)       DTWHWS    ( OUT RETURN_CODE )
```

ENVIRONMENT statement can vary on each operating systems; for example OS/390 differs slightly for SQL and ODBC access:

```
ENVIRONMENT (DTW_SQL)      DTWSQL    ( IN LOCATION, DB2SSID, DB2PLAN,
TRANSACTION_SCOPE)

ENVIRONMENT (DTW_ODBC)     DTWODBC   ( IN LOCATION, TRANSACTION_SCOPE)
```

Chapter 4. The Language Environment Programming Interface Utility Functions

Net.Data provides a programming interface for you to use when designing a new language environment. The language environment interface has utility functions that access Net.Data services that manage memory and configuration variables, and provide table and row manipulation features. “Appendix A. Language Environment Template” on page 71 provides a template that you can use as a model when designing your language environment.

The following section explains the Net.Data language environment interface utility functions.

Language Environment Utility Functions

Language environments use utility functions to access Net.Data services. These functions fall into four categories:

- “Utility Functions for Managing Memory”
- “Utility Functions for Managing Configuration Variables”
- “Utility Functions for Table Manipulation” on page 44
- “Utility Functions for Row Manipulation” on page 45

Utility Functions for Managing Memory

Language environments use the memory management utility functions to allocate storage owned by Net.Data, and to free storage that it allocated using the Net.Data run-time library.

The following example illustrates the need for these utility functions. Suppose that Net.Data is written using compiler A, with its corresponding run-time library. A programmer writes a new language environment, but uses compiler B, which has a different run-time library. The language environment cannot free storage that Net.Data allocated, and Net.Data cannot free storage that was allocated by the language environment because of potential incompatibilities between the two run-time libraries.

Table 5. Memory Management Utility Functions

Utility Function	Description
“dtw_malloc()” on page 48	Allocate storage from Net.Data’s run-time heap using dtw_malloc().
“dtw_free()” on page 46	Free storage allocated from Net.Data’s run-time heap using dtw_malloc().
“dtw_strdup()” on page 51	Allocate storage from Net.Data’s run-time heap and copy the specified string into the allocated storage using dtw_malloc().

Utility Functions for Managing Configuration Variables

The management utility functions for the configuration variables let language environments access configuration information stored in the Net.Data initialization file. Using these functions, all language environments can share the Net.Data initialization file and use information in it for configuring language environments.

Table 6. Configuration Utility Functions

Utility Function	Description
"dtw_getvar()" on page 47	Retrieve the value of a configuration variable from the Net.Data initialization file.

Utility Functions for Table Manipulation

Use the table functions to manipulate any Net.Data macro table variables that are passed to the language environment.

Row and column numbers begin with one (1).

Table 7. Table Utility Functions

Utility Function	Description
"dtw_table_New()" on page 62	Create a table object.
"dtw_table_Delete()" on page 54	Delete a table object.
"dtw_table_SetCols()" on page 65	Set the width of a table and allocate storage for the column headers.
"dtw_table_GetV()" on page 58	Retrieve a table value.
"dtw_table_SetV()" on page 67	Set a table value.
"dtw_table_GetN()" on page 57	Retrieve a table column heading.
"dtw_table_SetN()" on page 66	Set a table column heading.
"dtw_table_Rows()" on page 64	Retrieve the current number of rows in a table.
"dtw_table_Cols()" on page 53	Retrieve the current number of columns in a table.
"dtw_table_MaxRows()" on page 61	Retrieve the maximum allowable number of rows in a table.
"dtw_table_QueryColnoNj()" on page 63	Retrieve the column number of a column.
"dtw_table_AppendRow()" on page 52	Add one or more rows to the end of a table.
"dtw_table_InsertRow()" on page 60	Insert one or more rows in a table.
"dtw_table_DeleteRow()" on page 56	Delete one or more rows from a table.
"dtw_table_InsertCol()" on page 59	Insert one or more columns in a table.
"dtw_table_DeleteCol()" on page 55	Delete one or more columns from a table.

Utility Functions for Row Manipulation

The row utility functions manipulate the row object that is passed to a language environment's dtw_getNextRow() interface function during row-at-a-time processing.

| Row numbers begin with one (1).

Table 8. Row Utility Functions

Utility Function	Description
"dtw_row_SetCols()" on page 49	Set the width of a row.
"dtw_row_SetV()" on page 50	Set a table value.

Utility Functions Syntax Reference

This section describes each of the utility functions, their format, usage, and parameters, as well as providing a simple example.

dtw_free()

Usage

Frees storage that was allocated from Net.Data's run-time heap using dtw_malloc(). The buffer points to the allocated storage to free.

Format

```
void dtw_free(void *buffer)
```

Parameters

buffer	A pointer to the allocated storage to free.

Examples

```
char *myBuf;
long  nbytes = 8192;

myBuf = (char *)dtw_malloc(nbytes);

dtw_free((void *)myBuf);
```

dtw_getvar()

Usage

Retrieves the value of a configuration variable specified by *var_name* from the Net.Data initialization file. Net.Data owns the memory returned by dtw_getvar(); do not modify or free it.

Format

```
char *dtw_getvar(char *var_name)
```

Parameters

var_name	The name of the configuration variable to retrieve.

Examples

```
char *myBindFile;

myBindFile = dtw_getvar("BIND_FILE");
```

dtw_malloc()

Usage

Returns a pointer to storage that was allocated from Net.Data's run-time heap using dtw_malloc(). The storage is *nbytes* long. If Net.Data cannot return the requested storage, it returns a NULL pointer.

Format

```
void *dtw_malloc(long nbytes)
```

Parameters

nbytes	The number of bytes to allocate.

Examples

```
char *myBuf;
long  nbytes = 8192;

myBuf = (char *)dtw_malloc(nbytes);
```

dtw_row_SetCols()

Usage

Assigns the width of the row and allocates storage for the column headings. You can use the dtw_row_SetCols() utility function once for each row.

Format

```
int dtw_row_SetCols(void *row, int cols)
```

Parameters

row	A pointer to a newly created row which has not yet allocated any columns.
cols	The initial number of columns to allocate in the new row.

Examples

```
void *myRow;

rc = dtw_row_SetCols(myRow, 5);
```

dtw_row_SetV()

Usage

Assigns a table value. The caller of the dtw_row_SetV() utility function retains ownership of the memory pointed to by *src*. To delete the current table value, assign the value to NULL.

Format

```
int dtw_row_SetV(void *row, char *src, int col)
```

Parameters

row	A pointer to the row to modify.
src	A character string containing the new value to set.
col	The column number of the value to set.

Examples

```
void *myTable;
char *myFieldValue = "newValue";

rc = dtw_row_SetV(myRow, myFieldValue, 3);
```

dtw_strdup()

Usage

Allocates storage from Net.Data’s run-time heap and copies the string specified by *string* into the allocated storage using dtw_malloc(). If Net.Data cannot return the requested storage, it returns a NULL pointer.

Format

```
char *dtw_strdup(char *string)
```

Parameters

string	A pointer to the string value to copy into the storage allocated.

Examples

```
char *myString = "This string will be duplicated.";
char *myDupString;

myDupString = dtw_strdup(myString);
```

dtw_table_AppendRow()

Usage

Adds one or more rows to the end of the table. Assign the table values of the new rows with the dtw_table_SetV() utility after rows are appended to the table.

Format

```
int dtw_table_AppendRow(void *table, int rows)
```

Parameters

table	A pointer to the table to be appended to.
rows	The number of rows to append.

Examples

```
void *myTable;

rc = dtw_table_AppendRow(myTable, 10);
```

dtw_table_Cols()

Usage

Returns the current number of columns in the table.

Format

`int dtw_table_Cols(void *table)`

Parameters

table	A pointer to the table whose current number of columns is returned.

Examples

```
void *myTable;
int currentColumns;

currentColumns = dtw_table_Cols(myTable);
```

dtw_table_Delete()

Usage

Deletes all of the column headings, table values, and the table object.

Format

```
int dtw_table_Delete(void *table)
```

Parameters

table	A pointer to the table to delete.

Examples

```
void *myTable;

rc = dtw_table_Delete(myTable);
```

dtw_table_DeleteCol()

Usage

Deletes one or more columns beginning at the column specified in *start_col*. To delete all of the rows and columns of a table, substitute the utility function dtw_table_Cols() for the *cols* parameter.

```
dtw_table_DeleteCol(table, 1, dtw_table_Cols());
```

Format

```
int dtw_table_DeleteCol(void *table, int start_col, int cols)
```

Parameters

table	A pointer to the table to modify.
start_col	The column number of the first column to delete.
rows	The number of columns to delete.

Examples

```
void *myTable;

rc = dtw_table_DeleteCol(myTable, 1, 10);
```

dtw_table_DeleteRow()

Usage

Deletes one or more rows beginning at the row specified in *start_row*.

Format

```
int dtw_table_DeleteRow(void *table, int start_row, int rows)
```

Parameters

table	A pointer to the table to modify.
start_row	The row number of the first row to delete.
rows	The number of rows to delete.

Examples

```
void *myTable;

rc = dtw_table_DeleteRow(myTable, 3, 10);
```

dtw_table_GetN()

Usage

Retrieves a column heading. Net.Data owns the memory pointed to by *dest*; do not modify or free it.

Format

```
int dtw_table_GetN(void *table, char **dest, int col)
```

Parameters

table	A pointer to the table from which a column heading is retrieved.
dest	A pointer to the character string to contain the column heading.
col	The column number of the column heading.

Examples

```
void *myTable;
char *myColumnHeading;

rc = dtw_table_GetN(myTable, &myColumnHeading, 5);
```

dtw_table_GetV()

Usage

| Retrieves a value from a table. Net.Data owns the memory pointed to by *dest*; do
| not modify or free it.

Format

`int dtw_table_GetV(void *table, char **dest, int row, int col)`

Parameters

table	A pointer to the table from which a value is retrieved.
dest	A pointer to the character string that is to contain the value.
row	The row number of the value to retrieve.
col	The column number of the value to retrieve.

Examples

```
void *myTable;
char *myTableValue;

rc = dtw_table_GetV(myTable, &myTableValue, 3, 5);
```

dtw_table_InsertCol()

Usage

Inserts one or more columns after the specified column.

Format

`int dtw_table_InsertCol(void *table, int after_col, int cols)`

Parameters

table	A pointer to the table to modify.
after_col	The number of the column after which the new columns are to be inserted. To insert columns at the beginning of the table, specify 0.
cols	The number of columns to insert.

Examples

```
void *myTable;

rc = dtw_table_InsertCol(myTable, 3, 10);
```

dtw_table_InsertRow()

Usage

Inserts one or more rows after the specified row.

Format

`int dtw_table_InsertRow(void *table, int after_row, int rows)`

Parameters

table	A pointer to the table to modify.
after_row	The number of the row after which the new rows are inserted. To insert rows at the beginning of the table, specify 0.
rows	The number of rows to insert.

Examples

```
void *myTable;

rc = dtw_table_InsertRow(myTable, 3, 10);
```

dtw_table_MaxRows()

Usage

| Returns the maximum number of rows allowed for the Net.Data table as defined by the dtw_table_New() utility function's parameter, *row_lim*.

Format

```
int dtw_table_MaxRows(void *table)
```

Parameters

table	A pointer to the table from which the maximum number of rows is returned.

Examples

```
void *myTable;
int maximumRows;

maximumRows = dtw_table_MaxRows(myTable);
```

dtw_table_New()

Usage

Creates a Net.Data table object and initializes all column headings and field values to NULL. The caller specifies the initial number of rows and columns, and the maximum number of rows. If the initial number of rows and columns is 0, you must use the dtw_table_SetCols() function to specify the number of fields in a row before any table function calls.

Format

```
int dtw_table_New(void **table, int rows, int cols, int row_lim)
```

Parameters

table	The name of the new table.
rows	The initial number of rows to allocate in the new table.
cols	The initial number of columns to allocate in the new table.
row_lim	The maximum number of rows this table can contain.

Examples

```
void *myTable;

rc = dtw_table_New(&myTable, 20, 5, 100);
```

dtw_table_QueryColnoNj()

Usage

Returns the column number associated with a column heading.

Format

```
int dtw_table_QueryColnoNj(void *table, char *name)
```

Parameters

table	A pointer to the table to query.
name	A character string specifying the column heading for which the column number is returned. If the column heading does not exist in the table, 0 is returned.

Examples

```
void *myTable;
int columnNumber;

columnNumber = dtw_table_QueryColnoNj(myTable, "column 1");
```

dtw_table_Rows()

Usage

Returns the current number of rows in the table.

Format

`int dtw_table_Rows(void *table)`

Parameters

table	A pointer to the table whose current number of rows is returned.

Examples

```
void *myTable;
     int currentRows;

     currentRows = dtw_table_Rows(myTable);
```

dtw_table_SetCols()

Usage

Sets the number of columns of the table and allocates storage for the column headings. Specify the column headings when the table is created; otherwise, you must specify them by calling this utility function before using any other table functions. You can only use the dtw_table_SetCols() utility function once for a table. Afterwards, use the dtw_table_DeleteCol() or dtw_table_InsertCol() utility functions.

Format

```
int dtw_table_SetCols(void *table, int cols)
```

Parameters

table	A pointer to a new table that has no columns or rows allocated.
cols	The initial number of columns to allocate in the new table.

Examples

```
void *myTable;

rc = dtw_table_SetCols(myTable, 5);
```

dtw_table_SetN()

Usage

| Assigns a name to a column heading. The caller of the dtw_table_SetN() utility function retains ownership of the memory pointed to by the *src* parameter. To delete the column heading, assign the column heading value to NULL.

Format

```
int dtw_table_SetN(void *table, char *src, int col)
```

Parameters

table	A pointer to the table whose column heading is assigned.
src	A character string being assigned to the new column heading.
col	The number of the column.

Examples

```
void *myTable;
char *myColumnHeading = "newColumnHeading";

rc = dtw_table_SetN(myTable, myColumnHeading, 5);
```

dtw_table_SetV()

Usage

Assigns a value in a table. The caller of the dtw_table_SetV() utility function retains ownership of the memory pointed to by the *src* parameter. To delete the table value, assign the value to NULL.

Format

```
int dtw_table_SetV(void *table, char *src, int row, int col)
```

Parameters

table	A pointer to the table whose value is being assigned.
src	A character string assigned to the new value.
row	The row number of the new value.
col	The column number of the new value.

Examples

```
void *myTable;
char *myTableValue = "newValue";

rc = dtw_table_SetV(myTable, myTableValue, 3, 5);
```

Part 3. Appendixes

Appendix A. Language Environment Template

Use this template to create your own language environments.

```
/**********************************************************************/
/*                                                                    */
/* File Name                                                          */
/*                                                                    */
/* Description                                                        */
/*                                                                    */
/* Functions                                                          */
/*                                                                    */
/* Entry Points                                                       */
/*                                                                    */
/* Change Activity                                                    */
/*                                                                    */
/* Flag    Reason      Date      Developer     Description            */
/* ------  ----------  --------  ------------  ---------------------- */
/*                                                                    */
/**********************************************************************/

/*--------------------------------------------------------------------*/
/* Includes                                                           */
/*--------------------------------------------------------------------*/
#include "dtwle.h"
```

Figure 2. Language Environment Template (Part 1 of 14)

```
#ifdef __MVS__
#pragma export(dtw_initialize)
#pragma export(dtw_execute)
#pragma export(dtw_getNextRow)
#pragma export(dtw_cleanup)
#endif

#ifdef _AIX_
//*---------------------------------------------------------------------*/
/* Function                                                             */
/*    dtw_getFp                                                         */
/*                                                                      */
/* Purpose                                                              */
/*    Set function pointers to all Language Environment Interface      */
/*    routines being provided by this Language Environment. If a       */
/*    routine in the structure is not being provided, set that field   */
/*    to NULL.                                                          */
/*                                                                      */
/* Format                                                               */
/*    int dtw_getFp(dtw_fp_t *func_pointer)                            */
/*                                                                      */
/* Parameters                                                           */
/*    func_pointer     A pointer to a structure which will contain     */
/*                     function pointers for all functions provided    */
/*                     by this language environment.                   */
/*                                                                      */
/* Returns                                                              */
/*    Success ....... 0                                                 */
/*    Failure ....... -1                                                */
/*---------------------------------------------------------------------*/
int dtw_getFp(dtw_fp_t *func_pointer)
{
    func_pointer->dtw_initialize_fp = dtw_initialize;
    func_pointer->dtw_execute_fp = dtw_execute;
    func_pointer->dtw_getNextRow_fp = dtw_getNextRow;
    func_pointer->dtw_cleanup_fp = dtw_cleanup;
    return 0;
}
#endif
```

Figure 2. Language Environment Template (Part 2 of 14)

```
/*-------------------------------------------------------------------*/
/*                                                                   */
/* Function                                                          */
/*    dtw_initialize                                                 */
/*                                                                   */
/* Purpose                                                           */
/*                                                                   */
/* Format                                                            */
/*    int dtw_initialize(dtw_lei_t *le_interface)                    */
/*                                                                   */
/* Parameters                                                        */
/*    le_interface     A pointer to a structure containing the       */
/*                     following fields:                             */
/*                                                                   */
/*      function_name                                                */
/*      flags                                                        */
/*      exec_statement                                               */
/*      parm_data_array                                              */
/*      default_error_message                                        */
/*      le_opaque_data                                               */
/*      row                                                          */
/*                                                                   */
/* Returns                                                           */
/*    Success ....... 0                                              */
/*    Failure ....... 0                                              */
/*-------------------------------------------------------------------*/
int dtw_initialize(dtw_lei_t *le_interface)
{
    return rc;
}
```

Figure 2. Language Environment Template (Part 3 of 14)

```
/*--------------------------------------------------------------------*/
/*                                                                    */
/* Function                                                           */
/*    dtw_execute                                                     */
/*                                                                    */
/* Purpose                                                            */
/*                                                                    */
/* Format                                                             */
/*    int dtw_execute(dtw_lei_t *le_interface)                        */
/*                                                                    */
/* Parameters                                                         */
/*    le_interface     A pointer to a structure containing the        */
/*                     following fields:                              */
/*                                                                    */
/*      function_name                                                 */
/*      flags                                                         */
/*      exec_statement                                                */
/*      parm_data_array                                               */
/*      default_error_message                                         */
/*      le_opaque_data                                                */
/*      row                                                           */
/*                                                                    */
/* Returns                                                            */
/*    Success ....... 0                                               */
/*    Failure ....... 0                                               */
/*--------------------------------------------------------------------*/
int dtw_execute(dtw_lei_t *le_interface)
{
    /*----------------------------------------------------------------*/
    /* Determine if %exec statement was specified.                    */
    /*----------------------------------------------------------------*/
    if (le_interface->flags & DTW_STMT_EXEC) {
        /*------------------------------------------------------------*/
        /* Parse the %exec statement                                  */
        /*------------------------------------------------------------*/
        rc = processExecStmt(le_interface->exec_statement);
        if (rc)
          {
          }
      }
    else {
        /*------------------------------------------------------------*/
        /* Parse the inline data                                      */
        /*------------------------------------------------------------*/
        rc = processInlineData(le_interface->exec_statement);
        if (rc)
          {
          }
      }
```

Figure 2. Language Environment Template (Part 4 of 14)

```
    /*---------------------------------------------------------------*/
    /* Parse the input parameters                                    */
    /*---------------------------------------------------------------*/
    rc = processInputParms(le_interface->parm_data_array);
    if (rc)
      {
      }
    /*---------------------------------------------------------------*/
    /* Process the request                                           */
    /*---------------------------------------------------------------*/
    rc = processRequest();
    if (rc)
      {
      }
    /*---------------------------------------------------------------*/
    /* Process the output data                                       */
    /*---------------------------------------------------------------*/
    rc = processOutputParms(le_interface->parm_data_array);
    if (rc)
      {
      }
    /*---------------------------------------------------------------*/
    /* Process the return code and default error message             */
    /*---------------------------------------------------------------*/
    if (rc)
      {
        setErrorMessage(rc, &(le_interface->default_error_message));
      }
    /*---------------------------------------------------------------*/
    /* Cleanup and exit program.                                     */
    /*---------------------------------------------------------------*/
    return rc;
}
```

Figure 2. Language Environment Template (Part 5 of 14)

```
/*------------------------------------------------------------------*/
/*                                                                  */
/* Function                                                         */
/*    dtw_getNextRow                                                */
/*                                                                  */
/* Purpose                                                          */
/*                                                                  */
/* Format                                                           */
/*    int dtw_getNextRow(dtw_lei_t *le_interface)                   */
/*                                                                  */
/* Parameters                                                       */
/*    le_interface     A pointer to a structure containing the      */
/*                     following fields:                            */
/*                                                                  */
/*      function_name                                               */
/*      flags                                                       */
/*      exec_statement                                              */
/*      parm_data_array                                             */
/*      default_error_message                                       */
/*      le_opaque_data                                              */
/*      row                                                         */
/*                                                                  */
/* Returns                                                          */
/*    Success ....... 0                                             */
/*    Failure ....... 0                                             */
/*------------------------------------------------------------------*/
int dtw_getNextRow(dtw_lei_t *le_interface)
{
    return rc;
}
```

Figure 2. Language Environment Template (Part 6 of 14)

```
/*--------------------------------------------------------------------*/
/*                                                                    */
/* Function                                                           */
/*    dtw_cleanup                                                     */
/*                                                                    */
/* Purpose                                                            */
/*                                                                    */
/* Format                                                             */
/*    int dtw_cleanup(dtw_lei_t *le_interface)                        */
/*                                                                    */
/* Parameters                                                         */
/*    le_interface     A pointer to a structure containing the        */
/*                     following fields:                              */
/*                                                                    */
/*      function_name                                                 */
/*      flags                                                         */
/*      exec_statement                                                */
/*      parm_data_array                                               */
/*      default_error_message                                         */
/*      le_opaque_data                                                */
/*      row                                                           */
/*                                                                    */
/* Returns                                                            */
/*    Success ....... 0                                               */
/*    Failure ....... 0                                               */
/*--------------------------------------------------------------------*/
int dtw_cleanup(dtw_lei_t *le_interface)
{
    /*----------------------------------------------------------------*/
    /* Determine if this is normal or abnormal termination.           */
    /*----------------------------------------------------------------*/
    if (le_interface->flags & DTW_END_ABNORMAL) {
        /*------------------------------------------------------------*/
        /* Do abnormal termination cleanup.                           */
        /*------------------------------------------------------------*/
      }
    else {
        /*------------------------------------------------------------*/
        /* Do normal termination cleanup.                             */
        /*------------------------------------------------------------*/
      }

    return rc;
}
```

Figure 2. Language Environment Template (Part 7 of 14)

```
/*--------------------------------------------------------------------*/
/*                                                                    */
/* Function                                                           */
/*    processInputParms                                               */
/*                                                                    */
/* Purpose                                                            */
/*                                                                    */
/* Format                                                             */
/*    unsigned long processInputParms(dtw_parm_data_t *parm__data)   */
/*                                                                    */
/* Parameters                                                         */
/*    dtw_parm_data_t *parm_data                                      */
/*                                                                    */
/* Returns                                                            */
/*    Success ..... 0                                                 */
/*    Failure .....                                                   */
/*                                                                    */
/*--------------------------------------------------------------------*/
unsigned long processInputParms(dtw_parm_data_t *parm_data)
{
    /*----------------------------------------------------------------*/
    /* Loop through all the variables in the parameter data array.    */
    /* The array is terminated by a NULL entry, meaning the parm_name */
    /* field is set to NULL, the parm_value field is set to NULL, and */
    /* the parm_descriptor field is set to 0. However, the only valid */
    /* check for the end of the parameter data array is to check      */
    /* parm_descriptor == 0, since the parm_name field is NULL when a */
    /* literal string is passed in, and the parm_value field is set   */
    /* to NULL when an undeclared variable is passed in.              */
    /*----------------------------------------------------------------*/
    for (; parm_data->parm_descriptor != 0; ++parm_data) {
```

Figure 2. Language Environment Template (Part 8 of 14)

```
        /*------------------------------------------------------------*/
        /* Determine the usage of each input parameter.              */
        /*------------------------------------------------------------*/
        switch(parm_data->parm_descriptor & DTW_USAGE) {

            case(DTW_IN):
                /*----------------------------------------------------*/
                /* Determine the type of each input parameter.       */
                /*----------------------------------------------------*/
                switch (parm_data->parm_descriptor & DTW_TYPE) {
                    case DTW_STRING:
                        break;
                    case DTW_TABLE:
                        break;
                    default:
                        /*--------------------------------------------*/
                        /* Internal error - unknown data type         */
                        /*--------------------------------------------*/
                        break;
                  }
                break;

            case(DTW_OUT):
                break;

            case(DTW_INOUT):
                break;

            default:
                /*----------------------------------------------------*/
                /* Internal error - unknown usage                     */
                /*----------------------------------------------------*/
                break;
          }
      }
    return rc;
}
```

Figure 2. Language Environment Template (Part 9 of 14)

```
/*--------------------------------------------------------------------*/
/*                                                                    */
/* Function                                                           */
/*    processOutputParms()                                            */
/*                                                                    */
/* Purpose                                                            */
/*                                                                    */
/* Format                                                             */
/*    unsigned long processOutputParms(dtw_parm_data_t *parm_data)    */
/*                                                                    */
/* Parameters                                                         */
/*    dtw_parm_data_t *parm_data                                      */
/*                                                                    */
/* Returns                                                            */
/*    Success ........ 0                                              */
/*    Failure ........ -1                                             */
/*                                                                    */
/*--------------------------------------------------------------------*/
unsigned long processOutputParms(dtw_parm_data_t *parm_data) {
    /*----------------------------------------------------------------*/
    /* Get output data in some language environment-specific manner.  */
    /* This is entirely dependent on what the language environment    */
    /* is interfacing to, and how the LE chooses to interface to it.  */
    /*----------------------------------------------------------------*/
```

Figure 2. Language Environment Template (Part 10 of 14)

```
/    /*----------------------------------------------------------------*/
    /* Loop through all the parms in the parameter data array,        */
    /* looking for output parameters.                                 */
    /*----------------------------------------------------------------*/
    for (; parm_data->parm_descriptor != 0; ++parm_data) {

        /*------------------------------------------------------------*/
        /* Determine usage of each parameter.                         */
        /*------------------------------------------------------------*/
        if (pd_i->parm_descriptor & DTW_OUT) {
            /*--------------------------------------------------------*/
            /* Determine the type of each input parameter.            */
            /*--------------------------------------------------------*/
            switch (pd_i->parm_descriptor & DTW_TYPE) {
                case DTW_STRING:
                    /*------------------------------------------------*/
                    /* Give a string parameter a new value. If the    */
                    /* parameter value is not currently NULL, the     */
                    /* storage must be freed using an LE interface    */
                    /* utility function if it was allocated by        */
                    /* Net.Data.                                      */
                    /*------------------------------------------------*/
                    if (parm_data->parm_value != NULL)
                        dtw_free(parm_data->parm_value);
                    parm_data->parm_value = dtw_strdup(newValue);
                    break;
                case DTW_TABLE:
                    /*------------------------------------------------*/
                    /* Change the size of a table parameter. Use the  */
                    /* LE interface utility functions to modify the   */
                    /* table object.                                  */
                    /*------------------------------------------------*/
                    /*------------------------------------------------*/
                    /* First get the pointer to the table object.     */
                    /*------------------------------------------------*/
                    void *myTable = (void *) parm_data->parm_value;
```

Figure 2. Language Environment Template (Part 11 of 14)

```
/*----------------------------------------------*/
/* Next get the current size of the table.       */
/*----------------------------------------------*/
cols = dtw_table_Cols(myTable);
rows = dtw_table_Rows(myTable);
/*----------------------------------------------*/
/* Now set the new size (assumes the new size    */
/* values are valid).                            */
/*----------------------------------------------*/

/*----------------------------------------------*/
/* Set the columns first.                        */
/*----------------------------------------------*/
if (cols > newColValue)
  {
    dtw_table_DeleteCol(myTable,
                        newColValue + 1,
                        cols - newColValue);
  }
else if (cols < new_col_value)
  {
    dtw_table_InsertCol(myTable,
                        cols,
                        newColValue - cols);
  }

/*----------------------------------------------*/
/* Now set the rows.                             */
/*----------------------------------------------*/
if (newColValue > 0) {
    if (rows > newRowValue)
      {
        dtw_table_DeleteRow(myTable,
                            newRowValue + 1,
                            rows - newRowValue);
      }
    else if (rows < new_row_value)
      {
        dtw_table_InsertRow(myTable,
                            rows,
                            newRowValue - rows);
      }
  }
```

Figure 2. Language Environment Template (Part 12 of 14)

```
                    /*------------------------------------------------*/
                    /* Now get the last row/column value.             */
                    /*------------------------------------------------*/
                    dtw_table_GetV(myTable,
                                   &myValue;,
                                   newRowValue,
                                   newColValue);

                    /*------------------------------------------------*/
                    /* Delete the last row/column value.              */
                    /*------------------------------------------------*/
                    dtw_table_SetV(myTable,
                                   NULL,
                                   newRowValue,
                                   newColValue);

                    /*------------------------------------------------*/
                    /* Set the last row/column value.                 */
                    /*------------------------------------------------*/
                    dtw_table_SetV(myTable,
                                   dtw_strdup(myNewValue),
                                   newRowValue,
                                   newColValue);

                    break;
                default:
                    /*------------------------------------------------*/
                    /* Internal error - unknown data type             */
                    /*------------------------------------------------*/
                    break;
              }
          }
      }

  return 0;
}
```

Figure 2. Language Environment Template (Part 13 of 14)

```
/*------------------------------------------------------------------*/
/*                                                                  */
/* Function                                                         */
/*    setErrorMessage()                                             */
/*                                                                  */
/* Purpose                                                          */
/*                                                                  */
/* Format                                                           */
/*    unsigned long setErrorMessage(int returnCode,                 */
/*                                  char **defaultErrorMessage)     */
/*                                                                  */
/* Parameters                                                       */
/*    int    returnCode                                             */
/*    char **defaultErrorMessage                                    */
/*                                                                  */
/* Returns                                                          */
/*    Success ........ 0                                            */
/*    Failure ........ -1                                           */
/*                                                                  */
/*------------------------------------------------------------------*/
unsigned long setErrorMessage(int returnCode,
                             char **defaultErrorMessage)
{

    /*--------------------------------------------------------------*/
    /* Set the default error message based on the return code.      */
    /*--------------------------------------------------------------*/
    switch(returnCode) {
        case LE_SUCCESS:
            break;
        case LE_RC1:
            *defaultErrorMessage = dtw_strdup(LE_RC1_MESSAGE_TEXT);
            break;
        case LE_RC2:
            *defaultErrorMessage = dtw_strdup(LE_RC2_MESSAGE_TEXT);
            break;
        case LE_RC3:
            *defaultErrorMessage = dtw_strdup(LE_RC3_MESSAGE_TEXT);
            break;
        case LE_RC4:
            *defaultErrorMessage = dtw_strdup(LE_RC4_MESSAGE_TEXT);
            rc = LE_RC1INTERNAL;
            break;
      }
    return 0;
}
```

Figure 2. Language Environment Template (Part 14 of 14)

Appendix B. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, 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.

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 that has been exchanged, should contact:

IBM Corporation
555 Bailey Avenue, W92/H3
P.O. Box 49023
San Jose, CA 95161-9023

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement or any equivalent agreement between us.

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.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates 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. 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 IBM's application programming interfaces.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AIX
DataJoiner
DB2
Domino
IBM
IMS
Lotus
MVS
Net.Data
OS/2
OS/390
OS/400

The following terms are trademarks of other companies as follows:

Java and HotJava are trademarks of Sun Microsystems, Inc.

Microsoft, Windows, Windows NT®, and the Windows 95 logo are registered trademarks of Microsoft Corporation.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names may be trademarks or service marks of others.

Glossary

API. Application programming interface.

applet. A Java program included in an HTML page. Applets work with Java-enabled browsers, such as Netscape, and are loaded when the HTML page is loaded.

application programming interface (API). A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or licensed program. Net.Data supports the following proprietary Web server APIs for improved performance over CGI processes: ICAPI, GWAPI, ISAPI, and NSAPI.

BLOB. Binary large object.

cache. A type of memory that contains recently accessed data, designed to speed up subsequent access to the same data. The cache is often used to hold a local copy of frequently-used data that is accessible over a network.

caching. The processes of storing frequently-used results from a request to the Web server locally for quick retrieval, until it is time to refresh the information.

Cache Manager. The program that manages a cache for one machine. It can manage multiple caches.

CGI. Common Gateway Interface.

cliette. A long-running process that serves requests from the Web server. The Connection Manager schedules cliette processes to serve these requests.

CLOB. Character large object.

Common Gateway Interface. A standardized way for a Web server to pass control to an application program and receive data back.

Connection Manager. An executable file, `dtwcm`, in Net.Data that is needed to support Live Connection.

cookie. A packet of information sent by an HTTP server to a Web browser and then sent back by the browser each time it accesses that server. Cookies can contain any arbitrary information the server chooses and are used to maintain state between otherwise stateless HTTP transactions. *Free Online Dictionary of Computing*

database. A collection of tables, or a collection of table spaces and index spaces.

database management system (DBMS). A software system that controls the creation, organization, and modification of a database and access to the data stored within it.

data type. An attribute of columns and literals.

DBMS. Database management system.

firewall. A computer with software that guards an internal network from unauthorized external access.

flat file interface. A set of Net.Data built-in functions that let you read and write data from plain-text files.

HTML. Hypertext markup language.

HTTP. Hypertext transfer protocol.

hypertext markup language. A tag language used to write Web documents.

hypertext transfer protocol. The communication protocol used between a Web server and browser.

ICAPI. Internet Connection API.

ICS. Internet Connection Server.

ICSS. Internet Connection Secure Server.

Internet. An international public TCP/IP computer network.

Internet Connection Server. IBM's unsecure Web server.

Internet Connection Secure Server. IBM's secure Web server.

Intranet. A TCP/IP network inside a company firewall.

ISAPI. Microsoft's Internet Server API.

Java. An operating system-independent object-oriented programming language especially useful for Internet applications.

language environment. A module that provides access from a Net.Data macro to an external data source such as DB2 or a programming language such as Perl. Some language environments are supplied with Net.Data such as REXX, Perl, and Oracle. You can also create your own language environments.

Live Connection. A Net.Data configuration that works with the Connection Manager and Web server API. Live Connection enables database connections to be reused.

LOB. Large object.

middleware. Software that mediates between an application program and a network. It manages the interaction between disparate applications across the heterogeneous computing operating systems. *Free Online Dictionary of Computing*

NSAPI. Netscape API.

null. A special value that indicates the absence of information.

path. A search route used to locate files.

Perl. An interpreted programming language.

port. A 16-bit number used to communicate between TCP/IP and a higher-level protocol or application.

TCP/IP. Transmission Control Protocol / Internet Protocol.

Transmission Control Protocol / Internet Protocol. A set of communication protocols that support peer-to-peer connectivity functions for both local and wide-area networks.

URL. Uniform resource locator.

uniform resource locator. An address that names a HTTP server and optionally a directory and file name, for example: `http://www.software.ibm.com/data/net.data/index.html`.

Web server. A computer running http server software, such as Internet Connection.

Index

A
abnormal conditions
  dtw_lei flag 34, 39
  error messages 35

C
cleaning up
  after processing 37, 39
  dtw_lei flag 34, 39
  flag for abnormal conditions 34, 39
column headings
  allocating storage 49, 65
  assigning names 66
  deleting 54, 57, 66
  retrieving 57
  returning column number 63
columns
  deleting 55
  determining total in table 53
  inserting 59
  specifying number of in a table 65
configuration variables
  retrieving variable values 47
  utility functions for managing 43
configuring environments 39
creating tables 62

D
DB2, SQL language environment 23
dtw_ interface functions 36
dtw_ structures 33
dtw_ utilities 43
DTW_APPLET 9
DTW_FFI 5
DTW_IMS 8
DTW_JAVAPPS 15
DTW_LE_CONTINUE 38
dtw_lei
  fields
    default_error_messages 35
    exec_statement 35
    flags 34
    function name 34
    le_opaque_data 35
    parm_data_array 35
    row 35
  structure 34
DTW_ODBC 18
DTW_ORA 19
dtw_parm_data
  fields
    parm_descriptor 36
    parm_name 36
    parm_value 36
  structure 35
DTW_PERL 21
DTW_REXX 22
DTW_SQL 23
DTW_SYB 24
DTW_SYSTEM 26
DTW_WEBREG 26
dynamic memory allocation 36

E
ENVIRONMENT statements
  examples 41
  for new language environments 39
  syntax 39
error condition messages 35
error conditions 33
exec statements, dtw_lei flag 34
executing language environment statements 37

F
fatal errors, dtw_lei flag 34, 39
flat file interface
  built-in functions 6
  language environment 5
  security considerations 5
FUNCTION block
  executing statements 37
  name 34

G
glossary 86

H
heap, Net.Data run-time 36

I
IMS Web
  language environment 8
  Studio tool 8
initializing tasks, language environments 37
interface functions
  dtw_cleanup() 39
  dtw_execute() 37
  dtw_getNextRow() 38
  dtw_initialize() 37
  language environment, description 36
  processing order 37
invoking applets 9

J
Java applets
  classes 14
  creating 9
  example 13

Java applets *(continued)*
  generating tags 9
  invoking 9
  language environment 9
Java applications
  creating cliettes 17
  creating functions 16
  invoking 18
  Language Environment 15
  special configuration 17

L

language environments
  clean up after processing 37, 39
  configuring 39
  creating 31
  flat file interface 5
  IMS Web 8
  initializing 37
  interface functions 36
  interface template 71
  introduction 43
  Java applet 9
  Java applications 15
  ODBC 18
  Oracle 19
  Perl 21
  REXX 22
  SQL 23
  statements, executing 37
  structures 33
  summary 3
  Sybase 24
  system 26
  utility functions 43
  Web registry 26

M

maximum number of rows 61
memory management utility functions 43

N

Notices 85

O

ODBC
  language environment 18
  SQL statements in macro file 18
Oracle
  accessing 19
  language environment 19
  special configuration 19
  stored procedures 19

P

parameters
  naming 36
parameters *(continued)*
  parm_name 36
  passing 35, 36
  specifying 36
parm_data_array structures, assigning names 35
passing
  parameters 35
  variables 35
Perl
  language environment 21
  Net.Data variables in scripts 21
pointing to storage 48

R

REXX
  language environment 22
  variable substitution 23
row-at-a-time processing
  dtw_getNextRow() 37, 38
  DTW_LE_CONTINUE 34
  dtw_lei flag 34
row manipulation utility functions 45
rows
  appending 52
  assigning width 49
  deleting 55, 56
  dtw_getNextRow() interface function 35
  inserting 60
  retrieving current number of 64
  returning 35, 37, 38
  returning maximum allowed 61

S

SQL
  language environment 23
  supported statements 23
  using DataJoiner 23
storage
  allocating 48, 49, 51, 65
  dtw_lei flag 34
  freeing 34, 36, 46
structures, language environment
  dtw_lei 34
  dtw_parm_data 35
Sybase
  accessing 24
  language environment 24
  large objects 24
  special configuration 24
System
  language environment 26
  passing variables 26

T

table values
  assigning 50, 67
  deleting 54, 58, 67
  retrieving 58

tables
  appending rows 52
  creating new 62
  deleting 54
  manipulation utility functions 44
template, language environement 71

U

utility functions
  configuration variable 43
  dtw_free() 46
  dtw_getvar() 47
  dtw_malloc() 48
  dtw_row_SetCols() 49
  dtw_row_SetV() 50
  dtw_strdup() 51
  dtw_table_AppendRow() 52
  dtw_table_Cols() 53
  dtw_table_Delete() 54
  dtw_table_DeleteCol() 55
  dtw_table_DeleteRow() 56
  dtw_table_GetN() 57
  dtw_table_GetV() 58
  dtw_table_InsertCol() 59
  dtw_table_InsertRow() 60
  dtw_table_MaxRows() 61
  dtw_table_New() 62
  dtw_table_QueryColnoNj() 63
  dtw_table_Rows() 64
  dtw_table_SetCols() 65
  dtw_table_SetN() 66
  dtw_table_SetV() 67
  language environment 43
  memory management 43
  row manipulation 45
  table manipulation 44

V

variables
  freeing 39
  passing 35

W

Web registry
  description 27
  language environment 26

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.