IBM® Net.Data®
for OS/400

Administration and Programming Guide

IBM® Net.Data®
for OS/400

Administration and Programming Guide

Note

Be sure to read the information in “Notices” on page 157 before using this information and the product it supports.

October 2001 Edition

This document contains proprietary information of IBM. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this manual should not be interpreted as such.

Order publications through your IBM representative or the IBM branch office serving your locality or by calling 1-800-879-2755 in the United States or 1-800-IBM-4YOU in Canada.

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

This edition applies to:

- IBM HTTP Server for AS/400 (Program 5769-DG1), Version 4 Release 4 Modification 0
- IBM HTTP Server for iSeries (Program 5722-DG1), Version 5 Release 1 Modification 0

and to all subsequent versions, releases and modifications until otherwise indicated in new editions.

© Copyright International Business Machines Corporation 1997, 2001. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

© Copyright IBM Corp. 1997, 2001

Preface

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

About Net.Data

With Net.Data, you can create dynamic Web pages using data from both relational and non-relational database management systems (DBMSs), including DB2 databases that can be accessed through DRDA, and using applications written in programming languages such as Java, JavaScript, Perl, C, C++, and REXX.

Net.Data is a macro processor that executes as middleware on a Web server machine. 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, other data sources, 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 Navigator 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 applications written in, but not limited to, C, C++, RPG, COBOL, Java, or REXX programming languages.
- Access databases such as DB2
- Access other data sources such as flat files.

Net.Data passes this Web page to the Web server, which in turn forwards the page over the network for display at the browser.

Net.Data can be used in server environments that are configured to use interfaces such as HyperText Transfer Protocol (HTTP) and Common Gateway Interface (CGI). HTTP is an industry-standard interface for interaction between a browser and Web server, and CGI is an industry-standard interface for Web server invocation of gateway applications like Net.Data. Net.Data also supports a variety of Web server Application Programming Interfaces (APIs) for improved performance. The Net.Data family of products provide similar

© Copyright IBM Corp. 1997, 2001

capablities on the OS/400, OS/390, Windows NT, AIX, OS/2, HP-UX, Sun Solaris, Linux, and Dynix/PTX operating systems.

What's New in this Release?

Net.Data for OS/400 provides the following new features in this release:

- New flat file built-in functions: DTWF_COPY(), DTWF_EXISTS(), DTWF_WRITEFILE(), DTWF_READFILE.
- Flat file function DTWF_OPEN enhanced to allow files to be created in any CCSID.
- New built-in functions: DTW_EVAL, DTW_DATATYPE, DTW_PAD, DTW_ISNUMERIC.
- New tracing capabilities.
- New error logging capabilities.
- The ability to write user-specified messages to the Net.Data error log and the Net.Data trace log through built-in functions and user-written Language Environments.
- Ability to restrict file searches.
- Ability to override Net.Data error messages.

About This Book

This book discusses administration and programming concepts for Net.Data, as well as how to configure Net.Data and its components, plan for security, and improve performance.

Building on your knowledge of programming languages and database, you learn how to use the Net.Data macro language to develop macros. You learn how to use Net.Data-provided language environments that access DB2 databases, as well as using RPG, COBOL, and other programming languages to access your data.

This book may 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.ibm.com/software/data/net.data/

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

Who Should Read This Book

This book is intended for people involved in planning and writing Net.Data applications. To understand the concepts discussed in this book, you should

be familiar with how a Web server works, understand simple SQL statements, and know HTML tags, including HTML form tags.

The Net.Data macro language, variables, and built-in functions, as well as operating system differences are described in *Net.Data Reference*.

About Examples in This Book

Examples used in this book are kept simple to illustrate specific concepts and do not show every way Net.Data constructs can be used. Some examples are fragments that require additional code to work.

How to send your comments

Your feedback helps IBM to provide quality information. Please send any comments that you have about this book or other DB2 documentation. You can use any of the following methods to provide comments:

- Send your comments by e-mail to db2pubs@vnet.ibm.com and include the name of the product, the version number of the product, and the number of the book. If you are commenting on specific text, please list the location of the text (for example, a chapter and section title, page number, or a help topic title).
- Send your comments from the Web. Visit the Web site at:

  http://www.ibm.com/software/db2os390

  The Web site has a feedback page that you can use to send comments.
- Complete the readers' comment form at the back of the book and return it by mail, by fax (800-426-7773 for the United States and Canada), or by giving it to an IBM representative.
- Mail—Print and use the Readers' Comments form on the next page. To print the form, select **Print** or **Copy** from the **Services** pull-down menu. Enter *COMMENTS* as the topic to be printed or copied. Mail the completed form to:

  IBM Corporation, Department W92/H3
  P.O. Box 49023
  San Jose, CA 95161-9023
  U.S.A.

  If you are sending the form from a country other than the United States, give it to your local IBM branch office or IBM representative for mailing.
- Fax—Print and use the Readers' Comments form at the end of this book and fax it to this U.S. number: 800-426-7773 or (408) 463-4393. To print the form, follow the instructions under "Mail".

Chapter 1. Introduction

Net.Data is a server-side scripting language that extends Web servers by enabling the dynamic generation of Web pages using data from a variety of data sources. The data sources can include relational and non-relational database management systems such as DB2, DRDA-enabled databases, and flat file data. You can build applications rapidly using Net.Data's simple yet powerful cripting language. Net.Data allows reuse of existing business logic by supporting calls to applications written in a variety of programming languages, including Java, C/C++, RPG, CL, COBOL, REXX and others.

This chapter describes Net.Data and the reasons why you would choose to use it for your Web applications.

What is Net.Data?

Using Net.Data macros, you can execute programming logic, access and manipulate variables, call functions, and use report-generating tools. A macro is a text file containing Net.Data language constructs, which are used to build an application that can consist of HTML, XML, Javascript, and language environment statements, such as SQL. Net.Data processes the macro to produce output that can be displayed by a Web browser. Macros combine the simplicity of HTML with the dynamic functionality of Web server programs, making it easy to add live data to static Web pages. The live data can be extracted from local or remote databases and from flat files, or be generated by applications and system services.

Figure 1 on page 2 illustrates the relationship between Net.Data for OS/400, the Web server, and supported data and programming language environments.

© Copyright IBM Corp. 1997, 2001

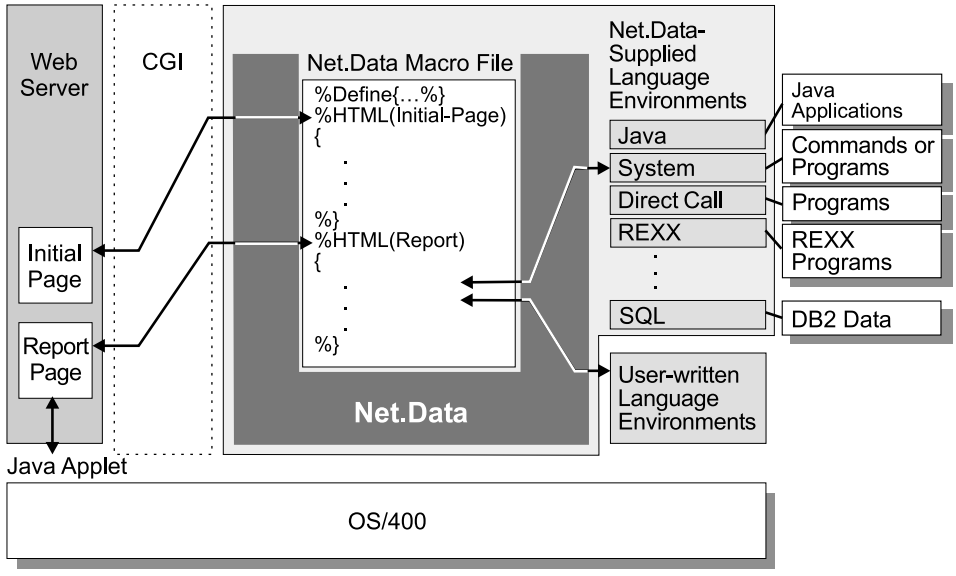

Figure 1. The Relationship between Net.Data for OS/400, the Web Server, and Supported Data and Program Sources

The Web server invokes Net.Data as a CGI application when it receives a URL that requests Net.Data services. The URL includes Net.Data-specific information, including the macro that is to be processed. When Net.Data finishes processing the request, it sends the resulting Web page to the Web server. The server passes it on to the Web client, where it is displayed by the browser.

Why Use Net.Data?

Net.Data is a good choice for creating dynamic Web pages because using the macro language is simpler than writing your own Web server applications and because Net.Data lets you use languages that you already know, such as HTML, SQL, REXX, and JavaScript. Net.Data also provides language environments that access DB2 databases, or use REXX, Perl, and other languages for your applications. In addition, changes to a macro can be seen instantaneously on a browser.

Net.Data complements data management capabilities that already exist on your operating system by enabling both data and related business logic for the Web. More specifically, Net.Data:

- Provides a simple, yet powerful macro language that allows for rapid development of Internet and Intranet applications.
- Permits the separation of data generation logic from presentation logic within your Web applications. Net.Data does not impose any restrictions on

the method with which the data is presented (such as HTML or Javascript). This separation allows users to easily change the presentation of data using the latest presentation techniques.

- Allows you to use existing skills and business logic to generate Web pages by providing the ability to interface with programs written in C, C++, RPG, COBOL, REXX, Java or other languages.
- Provides the ability to develop complex Internet applications quickly, using a simple macro language.
- Provides high-performance access to data that is stored in DB2 and in any remote DRDA-enabled database.
- Provides easy migration of macros between all operating systems supported by the Net.Data family of products.

Interpreted Macro Language

The Net.Data macro language is an interpreted language. When Net.Data is invoked to process a macro, Net.Data directly interprets each language statement in a sequential fashion, starting from the top of the file. Using this approach, any changes you make to a macro can be immediately seen when you next specify the URL that executes the macro. No recompilation is required.

Free Format

The Net.Data macro language has only a few rules about programming format. This simplicity provides programmers with freedom and flexibility. A single instruction can span many lines, or multiple instructions can be entered on a single line. Instructions can begin in any column. Spaces or entire lines can be skipped. Comments can be used anywhere.

Variables Without Type

Net.Data regards all data as character strings. Net.Data uses built-in functions to perform arithmetic operations on a string that represents a valid number, including those in exponential formats. Macro language variables are discussed in detail in “Net.Data Macro Variables” on page 61.

Built-in Functions

Net.Data supplies built-in functions that perform various processing, searching, and comparison operations for both text and numbers. Other built-in functions provide formatting capabilities and arithmetic calculations.

Error Handling

When Net.Data detects an error, messages with explanations are returned to the client. You can customize the error messages before

they are returned to a user at a browser. See “Configuration Variable Statements” on page 9 and the *Net.Data Reference* for more information.

Chapter 2. Configuring Net.Data

Net.Data for OS/400 is delivered as a standard part of:

- IBM TCP/IP Connectivity Utilities/400 V3R2, V3R7, V4R1, and V4R2
- IBM HTTP Server for AS/400 V4R3 and subsequent releases

There is nothing extra to buy; and there is no Net.Data software that you need to download and install.

The AS/400 TCP/IP and HTTP Server software that you need comes standard with OS/400, but is optionally installed. The following optional software should be installed on your system for the following versions of the OS/400 operating system:

- For IBM OS/400 operating system Version 3 Release 2, Version 3 Release 7, and subsequent versions and releases (57xx-SS1):
  - IBM TCP/IP Connectivity Utilities/400 (57xx-TC1)
- For IBM OS/400 operating system Version 4 Release 3, and subsequent versions and releases (57xx-SS1):
  - IBM HTTP Server for AS/400 (57xx-DG1)

After installing Net.Data, complete the steps described in the following sections to configure Net.Data for OS/400. The steps include:

- "Copying the Net.Data Program Object to Your CGI-BIN Library"
- "Creating an Initialization File" on page 8
- "Customizing the Net.Data Initialization File" on page 7
- "Setting Up Net.Data Language Environments" on page 27
- "Configuring the Web Server" on page 28
- "Granting Access Rights to Objects Accessed by Net.Data" on page 29

Copying the Net.Data Program Object to Your CGI-BIN Library

Before using Net.Data, you must copy the Net.Data program object to the CGI-BIN library and provide access rights to the object.

To copy the Net.Data program object:

1. Using the Create Duplicate Object (CRTDUPOBJ) command, copy the Net.Data program object, DB2WWW, from the QTCP library to a CGI-BIN library.

© Copyright IBM Corp. 1997, 2001

OS/400 V4R3 users: Use the program object in library QHTTPSVR; the program object in the QTCP library routes Net.Data requests to the QHTTPSVR library.

2. Change the DB2WWW program object in the CGI-BIN directory so that the user profile that CGI programs run under has access to the program object.

   By default, the DB2WWW program object authority for *PUBLIC users is set to *EXCLUDE. To provide access to the program object, change the program object's authority for *PUBLIC users to *USE, or specifically give the user profile access to the DB2WWW program object.

You can copy the Net.Data program object to multiple libraries for different applications. This allows you to have more than one version of the Net.Data initialization file or multiple protection schemes. See "Customizing the Net.Data Initialization File" on page 7 for more information about the Net.Data initialization file; see "Using Authentication" on page 36 for information on authentication.

To copy the Net.Data program object to multiple libraries:

1. Copy the Net.Data program object, DB2WWW, to a library using the steps listed above.
2. Associate the Net.Data program object with a CL program in each library.
   a. Create a CL program that calls the Net.Data program object located in the library specified in step 1.
   b. Copy the CL program to each library.

   In effect, the CL program you created becomes the Net.Data program object. If you do not associate the program object with a CL program, and copy the Net.Data program object DB2WWW to the different libraries, you get a -901 SQL code when using the SQL language environment.

   In the following sections, the CL program you created should be treated as the Net.Data program object, if you chose to create the CL program to call Net.Data.

About the Net.Data Initialization File

Net.Data uses its initialization file to establish the settings of various configuration variables and to configure language environments and search paths. The settings of configuration variables control various aspects of Net.Data operation, such as the following:

- Specifying a SMTP server and a character set for sending e-mail
- Enabling the SQL language environment variable SHOWSQL

The language environment statements define the Net.Data language environments that are available and identify special input and output parameter values that flow to and from the language environments. The language environments enable Net.Data to access different data sources, such as DB2 databases and system services. The path statements specify the directory paths to files that Net.Data uses, such as macros and programs.

Creating the Net.Data initialization file is optional with Net.Data for OS/400. By using an initialization file, you can use shorter URLs and shorter references to programs and include files within your Net.Data macros. However, you are required to have an initialization file if you decide to create your own language environment.

If you do not create an initialization file, Net.Data runs as if you have configured an initialization file with only the supported language environment statements (see “Chapter 6. Using Language Environments” on page 99 to learn about supported language environments). In this case, all macro, include, and executable references within the macro must be fully qualified.

Customizing the Net.Data Initialization File

The information contained in the initialization file is specified using three types of configuration statements, described in the following sections:

- “Configuration Variable Statements” on page 9
- “Path Configuration Statements” on page 20
- “Environment Configuration Statements” on page 24

See “Creating an Initialization File” on page 8 to learn how to create an initialization file.

The sample initialization file shown in Figure 2 on page 8 contains examples of these statements.

```
1  DTW_SMTP_SERVER 9.5.5.78
2  MACRO_PATH    /WWW/MACRO;/QSYS.LIB/WWW.LIB/MACRO.FILE
3  INCLUDE_PATH  /WWW/MACRO;/QSYS.LIB/WWW.LIB/MACRO.FILE
4  EXEC_PATH     /QSYS.LIB;/QSYS.LIB/WWW.LIB

5  ENVIRONMENT(MYLE1) /QSYS.LIB/LELIB.LIB/MYLE1.SRVPGM
   (IN VAR1, OUT VAR2)
```

- Line 1 sets the values of the configuration variables
- Lines 2- 4 define paths to the files that Net.Data needs to access
- Line 5 specifies a user-defined ENVIRONMENT statement

Figure 2. The Net.Data initialization file

The text of each individual configuration statement must all be on one line. Ensure that the initialization file contains an ENVIRONMENT statement for each user-defined language environment that you call from your macros. If you fully qualify all references to files within the macro, you do not need to specify any of the path configuration statements.

The following sections describe how to create the initialization file and customize the configuration statements in the initialization file.

- "Configuration Variable Statements" on page 9
- "Path Configuration Statements" on page 20

Creating an Initialization File

Creating an initialization file is optional when using Net.Data for OS/400. You should create an initialization file if:

- You want to set any of the Net.Data configuration variables to non-default values.
- You want to define the path statements for macro, include, and executable program files to shorten references to these files.
- You are using a language environment not supplied by Net.Data.

To create an initialization file:

1. In the library where the DB2WWW program object resides, use the Create Source Physical File (CRTSRCPF) command to create the initialization file.

   File name: INI

   Member name: DB2WWW

   It is recommended that you create the initialization file with a record length of 240 because the text of configuration statements must all be on one line.

2. Use the Source Entry Utility (SEU) or a workstation editor to add configuration statements to the file as demonstrated in the sample macro and in the following sections.

If you create an initialization file and then update it, you do not need to end or restart the Web server in order for the changes to take effect. Net.Data reads the initialization file once, during the initial invocation by an HTTP server job. The configuration data is saved so that on subsequent Net.Data invocations, Net.Data does not have to read the initialization file. However, if a change is made to the initialization file, Net.Data detects the change to the initialization file and reads the initialization file again.

Authorization Tip: Ensure that the user IDs under which Net.Data executes have the appropriate access rights to this file. See "Granting Access Rights to Objects Accessed by Net.Data" on page 29 for more information.

Configuration Variable Statements

Net.Data configuration variable statements set the values of configuration variables. Configuration variables are used for various purposes. Some variables are required by a language environment to work properly or to operate in an alternate mode. Other variables control the character encoding or content of the Web page being constructed. Additionally, you can use configuration variable statements to define application-specific variables.

The configuration variables you use depend on the language environments you are using, as well as other factors that are specific to the application.

To update the configuration variable statements:

Customize the initialization file with the configuration variables that are required for your application. A configuration variable has the following syntax:

NAME [=] *value-string*

The equal sign is optional, as denoted by the brackets.

The following sub-sections describe the configuration variables statements that you can specify in the initialization file:

DTW_DEFAULT_ERROR_MESSAGE: Specify Generic Error Messages

Use the DTW_DEFAULT_ERROR_MESSAGE configuration variable to specify a generic error message for applications in production. This variable provides a generic message for error conditions that are not captured in any MESSAGE block.

If you still wish to see the actual error messages generated by Net.Data, use error message logging to capture the messages. See to learn about using the error log.

If the configuration variable is not specified, Net.Data displays its own provided message for the error condition.

Syntax:

```
DTW_DEFAULT_ERROR_MESSAGE [=] "message"
```

Example: Specifies a generic message

```
DTW_DEFAULT_ERROR_MESSAGE "This site is temporarily unavailable."
```

DTW_ERROR_LOG_DIR: Location of Error Log

Sets the directory where the error log is stored. See “Chapter 9. Serviceability Features” on page 149 for more information on using this configuration variable.

Syntax:

```
DTW_ERROR_LOG_DIR [=] full_directory_path
```

DTW_ERROR_LOG_LEVEL: Level of Error to Log

Sets the level of error logging. See “Chapter 9. Serviceability Features” on page 149 for more information on using this configuration variable.

Syntax:

```
DTW_ERROR_LOG_LEVEL [=] OFF|INFORMATION|ERROR|INFORMATION+ERROR|ALL
```

Where:

OFF Specifies that no error messages are captured in the error message log. This is the default value.

INFORMATION
Net.Data will only log informational messages. For example, informational messages include messages that are issued when a URL is not valid, or when an HTML block is not found.

ERROR
Net.Data will only log error messages that are not handled by a message block.

INFORMATION+ERROR
Net.Data will log those messages that are logged by error log levels INFORMATION and ERROR.

ALL All messages are logged, including messages that are handled by message blocks.

Example:

```
DTW_ERROR_LOG_LEVEL ERROR
```

DTW_LOB_DIR

Specifies into which directory Net.Data writes large objects (LOBs).

The DTW_LOB_DIR must specify an IFS directory which is not in the QSYS.LIB file system.

Syntax:

```
DTW_LOB_DIR [=] path
```

Example: The following example shows the DTW_LOB_DIR configuration variable in the initialization file.

```
DTW_LOB_DIR /db2/lobs
```

When a query returns a LOB, Net.Data saves it in the directory specified in the DTW_LOB_DIR configuration variable.

Tip: Consider system limitations when using LOBs because they can quickly consume resources. See for more information.

DTW_MACRO_CACHE_SIZE: Macro Cache Size Variable

Indicates the memory size in megabytes that Net.Data should use when caching macros. When the cache size is exceeded, Net.Data removes old cached macros to make room in the cache. Net.Data removes the macros that have been used the least recently.

Syntax:

```
DTW_MACRO_CACHE_SIZE [=] size
```

Where:

size Specifies the cache memory size in number of megabytes. The default is 5 MB and caching is always enabled. If *size* is 0, no macros are cached. If *size* is 1 - 4, the default of 5 is used.

Example: Specifies a cache size of 16 MB.

```
DTW_MACRO_CACHE_SIZE 16
```

DTW_PAD_PGM_PARMS: Parameter Padding Configuration Variable

Indicates to a language environment whether character parameters that are to be passed to a program or stored procedure are padded with blanks. Character parameters have a data type of CHARACTER or CHAR.

For IN or INOUT parameters, if the length of parameter value is less than the specified precision, blanks are inserted to the right of the parameter value until the length of the parameter value is the same as the precision.

For OUT parameters, the parameter value is set to *precision* blanks.

After the call to the program or stored procedure, all trailing blanks are removed from OUT and INOUT parameter values.

Set this variable in the Net.Data initialization file to specify a value for all of your macros. You can override the value by defining it in the macro. If DTW_PAD_PGM_PARMS is not defined in the macro, it uses the value in the initialization file.

DTW_PAD_PGM_PARMS is supported by the Direct Call and SQL language environments.

Syntax:

```
DTW_PAD_PGM_PARMS [=] YES|NO
```

Where:

YES Specifies that all IN and INOUT character parameter values are left justified and padded with blanks for the defined precision of the parameter before passing the parameters to a program or stored procedure. Trailing blanks are removed after the call to a program or stored procedure.

NO Specifies that no padding is added to character parameter values (values are NULL-terminated) when passing parameters to programs or stored procedures. Trailing blanks are not removed after calling a program or stored procedure. This is the default.

DTW_REMOVE_WS: Variable for Removing Extra White Space

When this variable is set to YES, Net.Data removes extraenneous white space from the HTML output. By compressing white space, this variable reduces the amount of data sent to the Web browser, thereby improving performance. The default is NO.

You can override this variable in the macro by using the DEFINE statement.

Syntax:

```
DTW_REMOVE_WS [=] YES|NO
```

DTW_SHOWSQL: Enable or Disable SHOWSQL Configuration Variable

Overrides the effect of setting SHOWSQL within your Net.Data macros.

Syntax:

```
DTW_SHOWSQL [=] YES|NO
```

Where:

YES Enables SHOWSQL in any macro that sets the value of SHOWSQL to YES.

NO Disables SHOWSQL in your macros, even if the variable SHOWSQL is set to YES. NO is the default.

Table 1 on page 14 describes how the settings in the Net.Data initialization file and the macro determine whether the SHOWSQL variable is enabled or disabled for a particular macro.

Table 1. The Relationship Between Settings in the Net.Data Initialization File and the Macro for SHOWSQL

Setting of DTW_SHOWSQL	Setting SHOWSQL	SQL statement is displayed
NO	NO	NO
NO	YES	NO
YES	NO	NO
YES	YES	YES

DTW_SMTP_CCSID: E-mail SMTP CCSID Variable

Specifies the ASCII coded character set identifier (CCSID) associated with the Multi-purpose Internet Mail Extensions (MIME) character set specified in DTW_SMTP_CHARSET. The CCSID is to be used when translating data specified on the DTW_SENDMAIL function from EBCDIC to ASCII.

If DTW_SMTP_CCSID is specified, you must also specify DTW_SMTP_CHARSET. When specifying the CCSID, ensure that it is appropriate for the MIME character set specified in DTW_SMTP_CHARSET and that the CCSID is supported by the system. Table 2 on page 15 lists common MIME character sets and the associated ASCII CCSID. If DTW_SMTP_CCSID is not set, Net.Data uses the CCSID associated with MIME character set ISO-8859-1, which is 819.

Syntax:

```
DTW_SMTP_CCSID [=] ascii_ccsid
```

where *ascii_ccsid* is the ASCII CCSID (a number between 1-65534) to be used when translating from EBCDIC to ASCII.

Example:

```
DTW_SMTP_CCSID 912
```

This ASCII CCSID corresponds to the MIME character set ISO-8859-2

DTW_SMTP_CHARSET: E-mail SMTP Character Set Variable

Specifies the Multi-purpose Internet Mail Extensions (MIME) character set that is to be used in the e-mail messages by the DTW_SENDMAIL function. If DTW_SMTP_CHARSET is specified, you must also specify DTW_SMTP_CCSID. When specifying the MIME character set, ensure that the character set is valid because Net.Data does not validate the value specified for this variable. If DTW_SMTP_CHARSET is not set, Net.Data uses the MIME character set ISO-8859-1, with the associated CCSID of 819.

Table 2 lists common MIME character sets and the associated ASCII CCSID.

Table 2. Character sets supported by Net.Data

MIME Standard Character Set	ASCII CCSID	Description
US-ASCII	367	US English
ISO-2022-JP	5052	Japan MBCS
ISO-8859-1	819	Latin-1
ISO-8859-2	912	Latin-2
ISO-8859-5	915	Cyrillic
ISO-8859-6	1089	Arabic
ISO-8859-7	813	Greek
ISO-8859-8	916	Hebrew
ISO-8859-9	920	Latin-5

Syntax:

```
DTW_SMTP_CHARSET [=] character_set
```

Where *character_set* is the MIME character set to be used.

Example:

```
DTW_SMTP_CHARSET iso-8859-2
```

This MIME character set corresponds to the 912 ASCII CCSID.

DTW_SMTP_SERVER: E-mail SMTP Server Variable

Specifies the SMTP server to use for sending out e-mail messages using the DTW_SENDMAIL built-in function. The value of this variable can either be a host name or an IP address. If this variable is not set, Net.Data uses the local host as the SMTP server.

Syntax:

```
DTW_SMTP_SERVER [=] server_name
```

Where server_name is the host name or IP address of the the SMTP server that is to be used for sending e-mail messages.

Performance tip: Specify an IP address for this value to prevent Net.Data from connecting to a domain name server when retrieving the IP address of the specified SMTP server.

Example:

DTW_SMTP_SERVER 9.5.34.5

DTW_SQL_ISOLATION: DB2 Isolation Variable

The DTW_SQL language environment uses the DTW_SQL_ISOLATION configuration statement to determine the degree to which the database operations executed by the DTW_SQL language environment are isolated from concurrently executing processes.

Syntax:

DTW_SQL_ISOLATION *locking_method*

Where *locking_method* is one of the following values:

DTW_SQL_NO_COMMIT

Specifies not to use commitment control. For the OS/400 operating system, do not specify this value if a relational database is specified in the relational database directory and the relational database is on a non-OS/400 system.

DTW_SQL_READ_UNCOMMITTED

Specifies locking for the objects referred to in SQL ALTER, COMMENT ON, CREATE, DROP, GRANT, LABEL ON, and REVOKE statements and the rows updated, deleted, and inserted. The objects are locked until the end of the unit of work (transaction). Uncommitted changes in other processes can be seen.

DTW_SQL_READ_COMMITTED

Specifies locking for the objects referred to in SQL ALTER, COMMENT ON, CREATE, DROP, GRANT, LABEL ON, and REVOKE statements and the rows updated, deleted, and inserted. The objects are locked until the end of the unit of work (transaction). A row that is selected, but not updated, is locked until the next row is selected. Uncommitted changes in other processes cannot be seen.

DTW_SQL_REPEATABLE_READ

Specifies locking for the objects referred to in SQL ALTER, COMMENT ON, CREATE, DROP, GRANT, LABEL ON, and REVOKE statements and the rows selected, updated, deleted, and inserted. The objects are locked until the end of the unit of work (transaction). Uncommitted changes in other processes cannot be seen.

DTW_SQL_SERIALIZABLE

Specifies locking for the objects referred to in SQL ALTER, COMMENT ON, CREATE, DROP, GRANT, LABEL ON, and REVOKE statements and the rows selected, updated, deleted, and inserted. The objects are locked until the end of the unit of work (transaction). Uncommitted changes in other processes cannot be seen. All tables referred to in SELECT, UPDATE, DELETE, and INSERT statements are locked exclusively until the end of the unit of work (transaction).

DTW_SQL_NAMING_MODE: SQL Table Naming Variable

The DTW_SQL_NAMING_MODE configuration statement specifies how a table name can be specified in an SQL statement.

Syntax:

```
DTW_SQL_NAMING_MODE mode
```

Where *mode* is one of the following values:

SQL_NAMING

Specifies that tables are qualified by the collection name in the form:

collection.table

where *collection* is the name of the collection and *table* is the table name. The default qualifier is the user ID running the process that executes the SQL statement and is used when the table name is not explicitly qualified and the default collection name is not specified. SQL_NAMING is the default table name.

SYSTEM_NAMING

Specifies that files are qualified by library name in the form:

library/file

where *library* is the name of the library and *file* is the table name. The default search path is the library list (*LIBL) for the unqualified table name, if the table name (file) is not explicitly qualified and a default collection name (library) is not specified.

DTW_TRACE_LOG_DIR: Location of Trace File

Sets the directory where the trace log is stored. See “Chapter 9. Serviceability Features” on page 149 for more information on using this configuration variable.

Syntax:

```
DTW_TRACE_LOG_DIR [=] full_directory_path
```

Example:

```
DTW_TRACE_LOG_DIR /netdata/error/logs
```

DTW_TRACE_LOG_LEVEL: Level of Trace to Log

Sets the level of trace logging. See “Chapter 9. Serviceability Features” on page 149 for more information on using this configuration variable.

Syntax:

```
DTW_TRACE_LOG_LEVEL [=] OFF|APPLICATION|SERVICE
```

Where:

OFF Specifies that no trace data is captured in the trace log. This is the default value.

Application
Net.Data writes application-level trace messages to the trace log.

Service
Net.Data writes all trace messages to the trace log. This level of trace should only be used when asked to do so by IBM. The information in a SERVICE trace will not be helpful in debugging your own applications and will make reading the trace more difficult than necessary. You may be asked to set the level to SERVICE to help resolve product service questions.

Example:

```
DTW_TRACE_LOG_LEVEL APPLICATION
```

DTW_TRACE_MERGE_RECORDS: Merge Trace Records

Specifies whether trace records from all threads should be merged into a single trace log file. See "Chapter 9. Serviceability Features" on page 149 for more information on using this configuration variable.

Syntax:

```
DTW_TRACE_MERGE_RECORDS [=] YES|NO
```

Where:

YES Specifies that trace records are written to one file, NETDATA.TRACE. This is the default value

NO Specifies that trace records are written to thread-specific file. The file name is NETDATA.TRACE.xxxxx, where xxxxx is the process/thread identifier.

Example:

```
DTW_TRACE_MERGE_RECORDS  NO
```

DTW_UPLOAD_DIR

Specifies into which directory Net.Data will store files uploaded by the client. When this variable is not set, Net.Data will not accept the files for upload.

Syntax:

```
DTW_UPLOAD_DIR [=] path
```

Example:

```
DTW_UPLOAD_DIR /tmp/uploads
```

DTW_RESTRICT_PATH_SEARCH: Restrict Path Search

Specifies that Net.Data should restrict file searches to directories specified in MACRO_PATH, EXEC_PATH and INCLUDE_PATH configuration variables.

Syntax:

```
DTW_RESTRICT_PATH_SEARCH [=] YES|NO
```

Where:

YES Specifies that Net.Data will restrict file searches to directories specified in MACRO_PATH, EXEC_PATH and INCLUDE_PATH configuration variables.

NO Specifies that if Net.Data does not find a file in directories specified in MACRO_PATH, EXEC_PATH and INCLUDE_PATH configuration variables, Net.Data will attempt to use file as-is to determine if it exists. This is the default.

DTW_PROCESS_REPORT_ON_ERROR: Process Report On Error

Specifies whether the report block should be processed when an error occurs in a database function. This variable has been introduced in order to be compatible with Net.Data on other platforms, which do not process report blocks when an error occurs in a database function.

Syntax:

```
DTW_PROCESS_REPORT_ON_ERROR [=] YES|NO
```

Where:

YES Specifies that Net.Data will process the report block when an error occurs in a database function. This is the default.

NO Specifies that Net.Data will not process the report block when an error occurs in a database function.

DTW_JAVA_VMOPTIONS: Java Virtual Machine Options

Specifies Java options that is to be applied to the Java environment created by the Java language environment. The only Java option that Net.Data recognizes is java.version.

Syntax:

```
DTW_JAVA_VMOPTIONS [=] java-options
```

Where:

java-options
Specifies java options to be used.

Example:

```
Specify the java.version to use.
DTW_JAVA_VMOPTIONS  java.version=1.2
```

Path Configuration Statements

Net.Data determines the location of files and executable programs used by Net.Data macros from the settings of path configuration statements. The path statements are:

- "DTW_ATTACHMENT_PATH"
- "DTW_JAVA_CLASSPATH" on page 21
- "EXEC_PATH" on page 21
- "FFI_PATH" on page 22
- "INCLUDE_PATH" on page 22
- "MACRO_PATH" on page 23

These path statements identify one or more directories that Net.Data searches when attempting to locate macros, executable files, text files, and include files. The path statements that you need depend on the Net.Data capabilities that your macros use.

Update guidelines:

Some general guidelines apply to the path statements. Exceptions are noted in the description of each path statement.

- Seperate each specified directory in the path statement with a semicolon (;).
- Forward slashes (/) and back slashes (\) are treated the same.
- Each path statement can specify multiple paths. Paths are searched from left to right in the order specified. This multiple-path capability lets you organize your files within multiple directories. For example, you can place each of your Web applications in its own directory.
- It is recommended to use absolute path statements.

The following sections describe the purpose and syntax of each path statement and provide examples of valid path statements.

DTW_ATTACHMENT_PATH

This path configuration statement specifies the path used to locate attachments to be sent using DTW_SENDMAIL.

Syntax:

```
DTW_ATTACHMENT_PATH [=] path
```

Example:

```
DTW_ATTACHMENT_PATH /usr/lpp/internet/server_root/pub/upload
```

DTW_JAVA_CLASSPATH

This path configuration statement specifies the path used to locate Java classes. Directories are separated by colons.

Syntax:

```
DTW_JAVA_CLASSPATH [=] path
```

Example: The following example shows the DTW_JAVA_CLASSPATH statement in the initialization file.

Net.Data initialization file:

```
DTW_JAVA_CLASSPATH /directory1/directory2:/QIBM/ProdData/Java400
```

EXEC_PATH

This path configuration statement identifies one or more directories that Net.Data searches for an external program that is invoked by the EXEC statement or an executable variable. If the program is found, the external program name is appended to the path specification, resulting in a fully qualified file name that is passed to the language environment for execution.

Syntax:

```
EXEC_PATH [=] path1;path2;...;pathn
```

Example: The following example shows the EXEC PATH statement in the initialization file and the EXEC statement in the macro that invokes an external program.

Net.Data initialization file:

```
EXEC_PATH /qsys.lib/programs.lib;/qsys.lib/rexx.lib/rexxpgms.file;
```

Net.Data macro:

```
%FUNCTION(DTW_REXX) myFunction() {
  %EXEC{ myFunction.mbr %}
%}
```

If the file myFunction.mbr is found in the `/qsys.lib/rexx.lib/rexxpgms.file` directory, the qualified name of the program is `/qsys.lib/rexx.lib/rexxpgms.file/myFunction.mbr`.

If the file is not found in the directories specified in the EXEC_PATH statement:

- If the specified path is absolute, Net.Data searches for the file in the specified path. For example, if the following EXEC statement were specified:

```
%EXEC{/qsys.lib/programs.lib/rpg1.pgm %}
```

Net.Data would search for the file `rpg1.pgm` in the `/qsys.lib/programs.lib` directory.

- If the specified path is relative, Net.Data searches the current working directory. For example, if the following EXEC statement were specified:

  ```
  %EXEC { rpg1.pgm %}
  ```

  then Net.Data would attempt to find the file `rpg1.pgm` in the current working directory.

FFI_PATH

This path configuration statement identifies one or more directories that Net.Data searches for a flat file that is referenced by a flat file interface (FFI) function.

Syntax:

```
FFI_PATH [=] path1;path2;...;pathn
```

Example: The following example shows an FFI_PATH statement in the initialization file.

Net.Data initialization file:

```
FFI_PATH /u/user1/ffi;/usr/lpp/netdata/ffi;
```

When the FFI language environment is called, Net.Data looks in the path specified in the FFI_PATH statement.

Because the FFI_PATH statement is used to provide security to those files not in directories in the path statement, there are special provisions for FFI files that are not found. See the FFI built-in functions section in *Net.Data Reference*.

INCLUDE_PATH

This path configuration statement identifies one or more directories that Net.Data searches to find a file specified on an INCLUDE statement in a Net.Data macro. When it finds the file, Net.Data appends the include file name to the path specification to produce the qualified include file name.

Syntax:

```
INCLUDE_PATH [=] path1;path2;...;pathn
```

Example 1: The following example shows both the INCLUDE_PATH statement in the initialization file and the INCLUDE statement that specifies the include file.

Net.Data initialization file:

```
INCLUDE_PATH /u/user1/includes;/usr/lpp/netdata/includes
```

Net.Data macro:

```
%INCLUDE "myInclude.txt"
```

If the file *myInclude.txt* is found in the /u/user1/includes directory, the fully-qualified name of the include file is /u/user1/includes/myInclude.txt.

Example 2: The following example shows the INCLUDE_PATH statement and an INCLUDE file with a subdirectory name.

Net.Data initialization file:

```
INCLUDE_PATH /u/user1/includes;/usr/lpp/netdata/includes
```

Net.Data macro:

```
%INCLUDE "OE/oeheader.inc"
```

The include file is searched for in the directories /u/user1/includes/OE and /usr/lpp/netdata/includes/OE. If the file is found in /usr/lpp/netdata/includes/OE, the fully qualified name of the include file is /usr/lpp/netdata/includes/OE/oeheader.inc.

If the file is not found in the directories specified in the INCLUDE_PATH statement:

- If the specified path is absolute, Net.Data searches for the file in the specified path. For example, if the following INCLUDE statement were specified:

  ```
  %INCLUDE "/u/user1/includes/oeheader.inc
  ```

  then Net.Data would search for the file oeheader.inc in the /u/user1/includes directory.
- If the specified path is relative, Net.Data searches the current working directory. For example, if the following INCLUDE statement were specified:

  ```
  %INCLUDE "oeheader.inc"
  ```

  then Net.Data would attempt to find the file oeheader.inc in the current working directory.

MACRO_PATH

This path configuration statement identifies the directories that Net.Data searches for Net.Data macros. For example, specifying the following URL requests the Net.Data macro with the path and file name /macro/sqlm.dtw:

```
http://server/cgi-bin/db2www/macro/sqlm.dtw/report
```

Syntax:

```
MACRO_PATH [=] path1;path2;...;pathn
```

The equal sign (=) is optional, as indicated by brackets.

Net.Data appends the path /macro/sqlm.d2w/report to the paths in the MACRO_PATH configuration statement, from left to right until Net.Data finds the macro. If the macro is not found, Net.Data will execute the macro defined for the DTW_DEFAULT_MACRO configuration variable, or it will print an error. See "Chapter 4. Invoking Net.Data" on page 43 for information on invoking Net.Data macros.

Example: The following example shows the MACRO_PATH statement in the initialization file and the related link that invokes Net.Data.

Net.Data initialization file:

```
MACRO_PATH /u/user1/macros;/usr/lpp/netdata/macros
```

HTML link:

```
<a href="http://server/cgi-bin/db2www/query.dtw/input">Submit another query.</a>
```

If the file *query.dtw* is found in the directory /u/user1/macros, then the fully-qualified path is /u/user1/macros/query.dtw.

If the file is not found in the directories specified in the MACRO_PATH statement, Net.Data searches for the file in the root (/) directory. For example, if the following URL is submitted:

```
http://myserver/cgi-bin/db2www/myfile.txt/report
```

and the file myfile.txt was not found in any of the directories specified in MACRO_PATH, then Net.Data attempts to find the file in the root (/) directory:

```
/myfile.txt
```

Environment Configuration Statements

An ENVIRONMENT statement configures a language environment. A language environment is a component of Net.Data that Net.Data uses to access a data source such as a DB2 database or to execute a program written in a language such as REXX. Net.Data provides a set of language environments, as well as an interface that allows you to create your own language environments. These language environments are described in

"Chapter 6. Using Language Environments" on page 99 and the language environment interface is described in *Net.Data Language Environment Interface Reference*.

Net.Data requires that an ENVIRONMENT statement for a particular language environment exist before you can invoke that language environment.

Net.Data for OS/400 does not require an ENVIRONMENT statement for language environments that are shipped with Net.Data. However, if a language environment statement is encountered, it overrides the default that Net.Data uses. It is recommended that ENVIRONMENT statements for Net.Data supplied language environments not be added to the Net.Data configuration file.

You can associate variables with a language environment by specifying the variables as parameters in the ENVIRONMENT statement. Net.Data implicitly passes the parameters that are specified on an ENVIRONMENT statement to the language environment as macro variables. To change the value of a parameter that is specified on an ENVIRONMENT statement in the macro, either assign a value to the variable using the DTW_ASSIGN() function or define the variable in a DEFINE section.

Important: If a variable is defined in a macro but is not specified on the ENVIRONMENT statement, the macro variable will not be passed to the language environment.

For example, a macro can define a DATABASE variable to specify the name of a database at which an SQL statement within a DTW_SQL function is to be executed. The value of DATABASE must be passed to the SQL language environment (DTW_SQL) so that the SQL language environment can connect to the designated database. To pass the variable to the language environment, you must add the DATABASE variable to the parameter list of the environment statement for DTW_SQL.

The sample Net.Data initialization file makes several assumptions about customizing the setting of Net.Data environment configuration statements. These assumptions may not be correct for your environment. Modify the statements appropriately for your environment.

To add or update an ENVIRONMENT statement:

ENVIRONMENT statements have the following syntax:

```
ENVIRONMENT(type) library_name (parameter_list, ...)
```

Parameters:

- *type*

The name by which Net.Data associates this language environment with a FUNCTION block that is defined in a Net.Data macro. You must specify the type of the language environment on a FUNCTION block definition to identify the language environment that Net.Data should use to execute the function.

- *library_name*

  The name of the service program containing the language environment interfaces that Net.Data calls.

  The service program name is specified with the *.SRVPGM* extension.

- *parameter_list*

  The list of parameters that are passed to the language environment on each function call, in addition to the parameters that are specified in the FUNCTION block definition.

  To set and pass the variables in the parameters list, define the variable in the macro.

  You must define these parameters as configuration variables or as variables in your macro before executing a function that will be processed by the language environment. The following example specifies the variables in the ENVIRONMENT statement:

  ```
  ENVIRONMENT(DTW_SQL) /QSYS.LIB/QTCP.LIB/QTMHSQL.SRVPGM(IN
   DATABASE,LOGIN,PASSWORD,TRANSACTION_SCOPE)
  ```

  If a function modifies any of its output parameters, the parameters keep their modified value after the function completes.

When Net.Data processes the initialization file, it does not load the language environment service programs. Net.Data loads a language environment service program when it first executes a function that identifies that language environment. The service program then remains loaded for as long as Net.Data is loaded.

Example: ENVIRONMENT statements for Net.Data-provided language environments

When customizing the ENVIRONMENT statements for your application, add the variables to the ENVIRONMENT statements that need to be passed from your initialization file to a language environment or that Net.Data macro writers need to set or override in their macros.

On OS/400, ENVIRONMENT statements are not required for Net.Data language environments and are not recommended. However, this example shows some of the default ENVIRONMENT statements that Net.Data uses.

```
1  MACRO_PATH    /WWW/MACRO;/QSYS.LIB/WWW.LIB/MACRO.FILE
2  INCLUDE_PATH  /WWW/MACRO;/QSYS.LIB/WWW.LIB/MACRO.FILE
3  EXEC_PATH     /QSYS.LIB;/QSYS.LIB/WWW.LIB

4  ENVIRONMENT(DTW_REXX) /QSYS.LIB//QTCP.LIB/QTMHREXX.SRVPGM ( )
5  ENVIRONMENT(DTW_SQL)  /QSYS.LIB/QTCP.LIB/QTMHSQL.SRVPGM (IN DATABASE,
     LOGIN, PASSWORD, SHOWSQL, TRANSACTION_SCOPE, DB_CASE,
     RPT_MAX_ROWS, START_ROW_NUM, DTW_SET_TOTAL_ROWS,
     OUT DTWTABLE, SQL_CODE, TOTAL_ROWS)
6  ENVIRONMENT(DTW_SYSTEM) /QSYS.LIB/QTCP.LIB/QTMHSYS.SRVPGM  ( )
```

Required: Each ENVIRONMENT statement must be on a single line.

Setting Up Net.Data Language Environments

After you modify configuration variables and ENVIRONMENT configuration statements for the Net.Data language environments, some additional setup is required before the following language environments can function properly. The following sections describe the steps necessary to set up the language environments:

- "Setting up the Java Application Language Environment"
- "Setting up the SQL Language Environment"

Setting up the Java Application Language Environment

Before using the Java Application language environment, first introduced in OS/400 V4R4, complete the following steps:

1. Install the "AS/400 Developer Kit for Java" licensed program, product identifier 5769JV1. The "AS/400 Developer Kit for Java" must be installed to run Java applications on the AS/400.
2. Set the DTW_JAVA_CLASSPATH path configuration variable in the Net.Data initialization file so Java can find the Java application classes. For more information on this path configuration statement, see "DTW_JAVA_CLASSPATH" on page 21.
3. Set the DTW_JAVA_VMOPTIONS configuration variable to the JDK level that is to be used by the Java Application Language Environment. To learn more about this configuration variable, see "DTW_JAVA_VMOPTIONS: Java Virtual Machine Options" on page 19.

After setting up the Java Application language environment, see "Java Application Language Environment" on page 106 to learn how to use the Java Application language environment.

Setting up the SQL Language Environment

Before using the SQL language environment, complete the following steps:

1. Create a directory entry for the local database in the relational database directory (a directory entry with a remote location of *LOCAL), in addition to any remote databases that the SQL language environment needs to access.

   Add the entry by using the Add Relational Database Directory Entry (ADDRDBDIRE) command.

   If you are accessing a remote database, complete additional configuration steps, such as setting up communications between the local system and the remote system. For more information about distributed database support, see *OS/400 Distributed Database Programming*.
2. If you are using DataLinks, ensure that TCP/IP is configured on any systems that used, and that the DataLink File Manager is started and configured on all systems that will contain objects to be linked. For more information about DataLinks, see *DB2 for 0S/400 SQL Programming*
3. If large objects (LOBs) are going to be returned by the SQL language environment, set the DTW_LOB_DIR configuration variable. To learn more about this configuration variable, see "DTW_LOB_DIR" on page 11.
4. Add or update configuration variables. The SQL language environment supports the following configuration variables that can be specified in a Net.Data initialization file:

   DTW_SQL_ISOLATION
   Determines the degree to which the database operations executed by the SQL language environment are isolated from concurrently executing processes

   DTW_SQL_NAMING_MODE
   Determines how a table name can be specified in an SQL statement

   DTW_SHOWSQL
   Enables the use of the macro variable SHOWSQL

   To learn more about the Net.Data configuration variable statements, see "Configuration Variable Statements" on page 9.

After setting up the SQL language environment, see "SQL Language Environment" on page 113 to learn how to use the SQL language environment.

Configuring the Web Server

The Common Gateway Interface (CGI) is an industry-standard interface that enables a Web server to invoke an application program such as Net.Data. Net.Data's support for CGI lets you use Net.Data with your favorite Web server.

Configure Net.Data to use only one interface at a time. For example, if you configure the Web server to execute Net.Data using CGI, do not also configure the Web server to execute Net.Data using another interface. If you want to later run Net.Data using another interface, such as FastCGI, then reconfigure the Web server solely for the new interface.

Configure the Web server to invoke Net.Data by adding Map, Exec, and Pass directives to the HTTP configuration file so that Net.Data gets invoked.

For example, assuming the Net.Data program object resides in library CGI, then the following directives redirect Net.Data requests to /QSYS.LIB/CGI.LIB/DB2WWW.PGM:

```
Map /cgi-bin/db2www/* /QSYS.LIB/CGI.LIB/DB2WWW.PGM/*
Map /CGI-BIN/DB2WWW/* /QSYS.LIB/CGI.LIB/DB2WWW.PGM/*
Exec /QSYS.LIB/CGI.LIB/*
```

Recommendation: Organize the directives in the following order within the HTTP configuration file to prevent directives from being ignored: Map, Exec, Pass. For example, if the following Pass directive precedes a Map or Exec directive, the Map and Exec directives are ignored:

```
Pass /*
```

Map directives

The Map directives map entries using the format `/cgi-bin/db2www/*` to the library where the Net.Data program resides on your system. (The asterisk (*) at the end of the string refers to anything that follows the string.) Both upper- and lower-case map statements are included, because the directives are case sensitive. In this example, both Map statements point to the same location.

Exec directives

The Exec directive enables the Web server to execute any CGI programs in the CGI library. Specify the library where the program resides (not the program itself) on the directive.

Granting Access Rights to Objects Accessed by Net.Data

Before using Net.Data, you need to ensure that the user IDs under which Net.Data executes have the appropriate access rights to objects that are referenced in a Net.Data macro and to the macro that a URL references.

More specifically, ensure that the user IDs under which Net.Data executes have the following authorizations:

- To read the Net.Data initialization file, `INI.FILE/DB2WWW.MBR`

- To execute the Net.Data executable files and service programs, and to search the directories (libraries) in the paths to the executable files and service programs
- To read the appropriate Net.Data macros and search the appropriate directories identified by the MACRO_PATH path configuration statement
- To execute the appropriate files and to search the appropriate directories identified by the EXEC_PATH path configuration statement
- To read the appropriate files and to search the appropriate directories identified by the INCLUDE_PATH path configuration statement
- To read and write the appropriate files, and to search the appropriate directories identified by the FFI_PATH path configuration statement
- To access any object that might be referenced by the target of a language environment statement. For example SQL language environment runs SQL statements, and SQL statements access database files, so the user ID that Net.Data is running under must have authority to the database files.

Examples:

Depending on the file system in which you choose to store your Net.Data macros, you need to authorize the user profile under which the Net.Data CGI program is run to the Net.Data macro. The following methods give the QTMHHTP1 user profile authority (in V3R2 and V3R7, Internet Connection for AS/400 ran CGI programs only under the QTMHHTP1 user profile.):

- In the root file system, use the Change Authority (CHGAUT) CL command to give authority to the user profile:

```
CHGAUT OBJ('/WWW') USER(QTMHHTP1) DTAAUT(*RX)
CHGAUT OBJ('/WWW/macro') USER(QTMHHTP1) DTAAUT(*RX)
CHGAUT OBJ('/WWW/macro/*') USER(QTMHHTP1) DTAAUT(*RX)
```

  You need to give authority to all objects in the path.

- In the library file system (QSYS.LIB), use the Grant Object Authority (GRTOBJAUT) CL command to give authority to the user profile:

```
GRTOBJAUT OBJ(WWW) OBJTYPE(*LIB) USER(QTMHHTP1) AUT(*USE)
GRTOBJAUT OBJ(WWW/MACRO) OBJTYPE(*FILE) USER(QTMHHTP1) AUT(*USE)
```

  You need to give authority only to the library and the source physical file.

  You can also use the CHGAUT CL command to give authority to objects in the QSYS.LIB file system as follows:

```
CHGAUT OBJ('/QSYS.LIB/WWW.LIB') USER(QTMHHTP1) DTAAUT(*RX)
CHGAUT OBJ('/QSYS.LIB/WWW.LIB/MACRO.FILE') USER(QTMHHTP1) DTAAUT(*RX)
```

Language environment-specific authority considerations are documented in each language environment section in “Chapter 6. Using Language Environments” on page 99.

Chapter 3. Keeping Your Assets Secure

Internet security is provided through a combination of firewall technology, operating systems features, Web server features, Net.Data mechanisms, and the access control mechanisms that are part of your data sources.

You must decide on what level of security is appropriate for your assets. This chapter describes methods you can use for keeping your assets secure and also provides references to additional resources you can use to plan for the security of your Web site.

The following sections contain guidelines for protecting your assets. The security mechanisms described include:

Using Firewalls

Firewalls are collections of hardware, software, and policies that are designed to limit access to resources in a networked environment.

Firewalls:

- Protect the internal network from infiltration or intrusion
- Protect the internal network from data and programs that are brought in by internal users
- Limit internal user access to external data
- Limit the damage that can be done if the firewall is breached

Net.Data can be used with firewall products that execute in your environment.

The following possible configurations provide recommendations for managing the security of your Net.Data application. These configurations provide high-level information and assume that you have configured a firewall that isolates your secure intranet from the public Internet. Carefully consider these configurations with your organization's security policies:

- **High security configuration**

© Copyright IBM Corp. 1997, 2001

This configuration creates a subnetwork that isolates Net.Data and the Web server from both the secure intranet and the public Internet. The firewall software is used to create a firewall between the Web server and the public Internet, and another firewall between the Web server and the secured intranet, which contains DB2 Server. This configuration is shown by Figure 3.

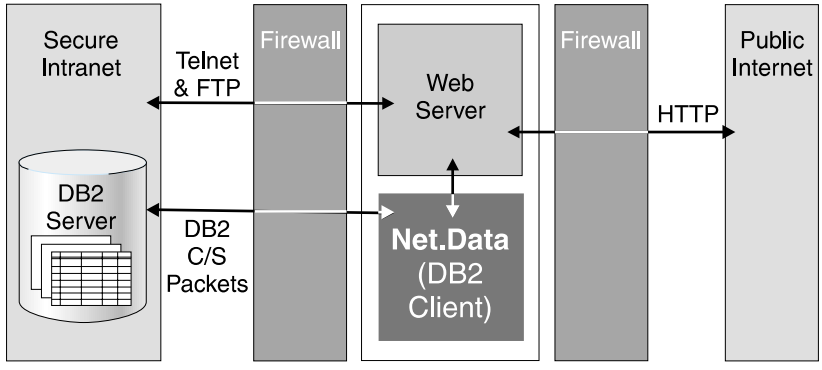

Figure 3. High Security Configuration

To set up this configuration:

- Install Net.Data on the Web server machine and ensure that Net.Data can access DB2 Server inside the intranet by configuring the firewall to allow DB2 traffic through the firewall. One method is to add a packet filtering rule to allow DB2 client requests from Net.Data and acknowledge packets from DB2 Server to Net.Data.
- Allow FTP and Telnet access between the Web server and the secure intranet. One method is to install a socks server on the Web server machine.
- In the packet filtering configuration file of the firewall software, specify that incoming TCP packets from the standard HTTP port can access the Web server. Also, specify that outgoing TCP acknowledge packets can go to any hosts on the public Internet from the Web server.

- **Intermediate security configuration**

  In this configuration, firewall software isolates the secured intranet with DB2 server from the public Internet. Net.Data and the Web server are outside the firewall on a workstation platform. This configuration is simpler than the first, but still offers database protection. Figure 4 on page 35 shows this configuration.

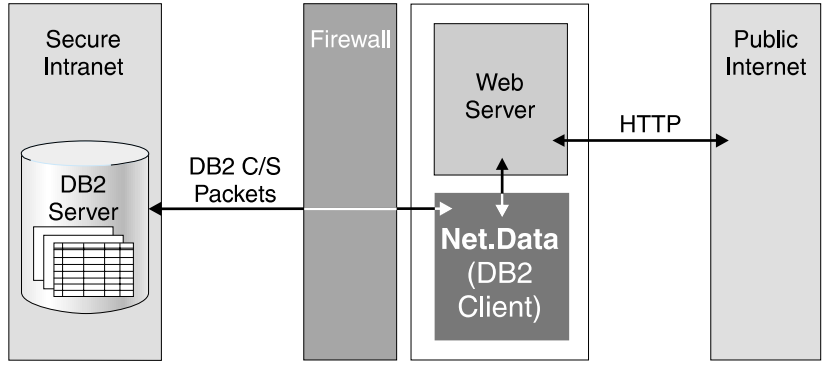

Figure 4. Intermediate Security Configuration:

The firewall must be configured to allow DB2 client requests to flow from Net.Data to DB2 and to allow acknowledge packets to flow from DB2 to Net.Data.

- **Low security configuration**

  In this configuration, DB2 server and Net.Data are installed outside of the firewall and the secured intranet. They are not protected from external attacks. The firewall needs no packet filtering rules for this configuration. Figure 5 shows this configuration.

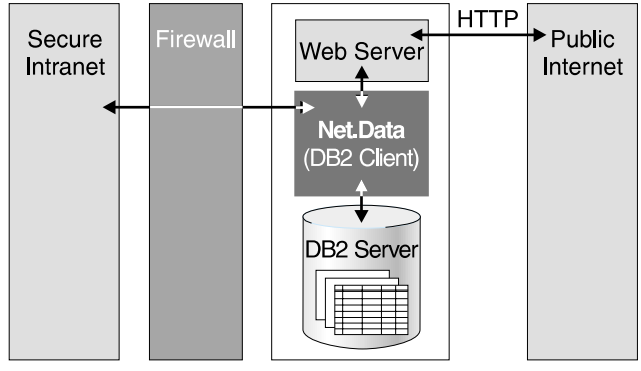

Figure 5. Low Security Configuration:

Encrypting Your Data on the Network

You can encrypt all data that is sent between a client system and your Web server when you use a Web server that supports Secured Sockets Layer (SSL). This security measure supports the encryption of login IDs, passwords, and all data that is transmitted through HTML forms from the client system to the Web server and all data that is sent from the Web server to the client system.

Using Authentication

Authentication is used to ensure that a user ID making a Net.Data request is authorized to access and update data within the application. Authentication is the process of matching the user ID with a password to validate that the request comes from a valid user ID. The Web server associates a user ID with each Net.Data request that it processes. The process or thread that is handling the request can then access any resource to which that user ID is authorized.

In an OS/400 environment, a user ID can become associated with the thread or process that is handling a Net.Data request in one of three ways:

Client-based authentication

The user is prompted for a local OS/400 user ID and password at the client. The Web server then authenticates the user. If successfully authenticated, the supplied user ID is associated with the request. Use of the special Web server %%CLIENT%% access control user ID enables this type of authentication.

Client-based authentication is supported by IBM's HTTP server starting in OS/400 V4R1.

Server-based authentication

The user ID of the Web server is associated with each request and the user is not prompted for a user ID or password. Use of the special Web server %%SERVER%% access control user ID enables this type of authentication.

By default, IBM's HTTP server runs CGI programs under the QTMHHTP1 user ID (user profile). However, if the UserID directive is in effect or within a protection setup where the UserID subdirective has been specified, the program is executed under the specified user ID.

Surrogate authentication

A surrogate user ID that has the authority to access some predefined collection of resources is associated with the client request. This type of authentication requires the creation of surrogate user IDs with access authority that is appropriate for a group of users or class of requests. Authentication with surrogate user IDs usually uses validation list objects first introduced in V4R1. For more information and examples, see *OS/400 System API Reference*.

The approach that the Web server uses for associating a user ID with a client request is specified when the Web server is configured. For additional detail on access control user IDs, on installing the Web server, and on using the Protect, Protection, DefProt, and UserId directives to configure the Web server, refer to your Web server documentation.

Tip: To protect Net.Data macros do the following:

1. Add protection directives in the Web server configuration file for the Net.Data program object.
2. Ensure the user ID that Net.Data will be running under has access rights to the macros. For more information on granting access rights, see "Granting Access Rights to Objects Accessed by Net.Data" on page 29.

Using Authorization

Authorization provides a user with complete or restricted access to an object, resource, or function. Data sources such as DB2 provide their own authorization mechanisms to protect the information that they manage. These authorization mechanisms assume that the user ID associated with the Net.Data request has been properly authenticated, as explained in "Using Authentication" on page 36. The existing access control mechanisms for these data sources then either permit or deny access based on the authorizations that are held by the authenticated user ID.

Using Net.Data Mechanisms

In addition to the methods described above, you can use Net.Data configuration variables or macro development techniques to limit the activities of end users, to conceal corporate assets such as the design of your database, and to validate user-provided input values within production environments.

Net.Data Configuration Variables

Net.Data provides several configuration variables that can be used to limit the activities of end users or conceal the design of your database.

Control file access with path statements

Net.Data evaluates the settings of path configuration statements to determine the location of files and executable programs that are used by Net.Data macros. These path statements identify one or more directories that Net.Data searches when attempting to locate macros, executable files, include files, or other flat files. By selectively including directories on these path statements, you can explicitly control the files that are accessible by users at browsers. Refer to "Chapter 2. Configuring Net.Data" on page 5 for additional detail about path statements.

You should also use authorization checking as described in "Using Authorization" and verify that file names cannot be changed in INCLUDE statements as described in "Macro Development Techniques" on page 38.

Disable SHOWSQL for production systems

The SHOWSQL variable allows the user to specify that Net.Data displays the SQL statements specified within Net.Data functions at a Web browser. This variable is used primarily for developing and testing the SQL within an application and is not intended for use in production systems.

You can disable the display of SQL statements in production environments using one of the following methods:

- When using versions of Net.Data that support the DTW_SHOWSQL configuration variable, use this variable in the Net.Data initialization file to override the effect of setting SHOWSQL within your Net.Data macros. See "DTW_SHOWSQL: Enable or Disable SHOWSQL Configuration Variable" on page 13 for syntax and additional information.
- Use the DTW_ASSIGN() function as described in "Macro Development Techniques".

See SHOWSQL in the variables chapter of *Net.Data Reference* for syntax and examples for the SHOWSQL Net.Data variable.

Macro Development Techniques

Net.Data provides several mechanisms that allow users to assign values to input variables. To ensure that macros execute in the manner intended, these input variables should be validated by the macro. Your database and application should also be designed to limit a user's access to the data that the user is authorized to see.

Use the following development techniques when writing your Net.Data macros. These techniques will help you ensure that your applications execute as intended and that access to data is limited to properly authorized users.

Ensure that Net.Data variables cannot be overridden in a URL

The setting of Net.Data variables by a user within a URL overrides the effect of DEFINE statements used to initialize variables in a macro. This might alter the manner in which your macro executes. To safeguard against this possibility, initialize your Net.Data variables using the DTW_ASSIGN() function.

Example: Instead of using:

```
%define START_ROW_NUM = "1"
```

Use:

```
@DTW_ASSIGN(START_ROW_NUM, "1")
```

Assigning the variable this way prevents a query string assignment such as ″START_ROW_NUM=10″ from overriding your macro setting.

Validate that your SQL statements cannot be modified in ways that alter the intended behavior of your application

Adding a Net.Data variable to an SQL statement within a macro allows users to dynamically alter the SQL statement before executing it. It is the responsibility of the macro writer to validate user-provided input values and ensure that an SQL statement containing a variable reference is not being modified in an unexpected manner. Your Net.Data application should validate user-provided input values from the URL so the Net.Data application can reject invalid input. Your validation design process should include for the following steps:

1. Identify the syntax of valid input; for example, a customer ID must start with a letter and can contain only alphanumeric characters.
2. Determine what potential harm can be caused by allowing incorrect input, intentionally harmful input, or input entered to gain access to internal assets of the Net.Data application.
3. Include input verification statements in the macro that meet the needs of the application. Such verification depends on the syntax of the input and how it is used. In simpler cases it can be enough to check for invalid content in the input or to invoke Net.Data to verify the type of the input. If the syntax of the input is more complex, the macro developer might have to parse the input partially or completely to verify whether it is valid.

Example 1: Using the DTW_POS() string function to verify SQL statements

```
%FUNCTION(DTW_SQL) query1() {
   select * from shopper where shlogid = '$(shlogid)'
%}
```

The value of the `shlogid` variable is intended to be a shopper ID. Its purpose is to limit the rows returned by the SELECT statement to rows that contain information about the shopper identified by the shopper ID. However, if the string "`smith' or shlogid<>'smith`" is passed as the value of the variable `shlogid`, the query becomes:

```
select * from shopper where shlogid = 'smith' or shlogid<>'smith'
```

This user-modified version of the original SQL SELECT statement returns the entire shopper table.

The Net.Data string functions can be used to verify that the SQL statement is not modified by the user in inappropriate ways. For example, the following logic can be used to ensure that single-quotes are not used to modify SQL statements:

```
@DTW_ADDQUOTE(shlogid, shlogid)
@query1()
```

The query then becomes:

```
select * from shopper where shlogid = 'smith'' or shlogid<>''smith'
```

Ensure that a file name in an INCLUDE statement is not modified in ways that alter the intended behavior of your application

If you specify the value for the file name with an INCLUDE statement using a Net.Data variable, then the file to be included is not determined until the INCLUDE file is executed. If your intent is to set the value of this variable within your macro, but to not allow a user at the browser to override the macro-provided value, then you should set the value of the variable using DTW_ASSIGN instead of DEFINE. If you do intend to permit the user at a browser to provide a value for the file name, then your macro should validate the value provided.

Example: A query string assignment such as `filename="../../x"` can result in the inclusion of a file from a directory not normally specified in the INCLUDE_PATH configuration statement. Suppose that your Net.Data initialization file contains the following path configuration statement:

```
INCLUDE_PATH /usr/lpp/netdata/include
```

and that your Net.Data macro contains the following INCLUDE statement:

```
%INCLUDE "$(filename)"
```

A query string assignment of `filename="../../x"` would include the file `/usr/lpp/x` , which was not intended by the INCLUDE_PATH configuration statement specification.

The Net.Data string functions can be used to verify that the file name provided is appropriate for the application. For example, the following logic can be used to ensure that the input value associated with the file name variable does not contain the string ″..″:

```
@DTW_POS("..", $(filename), result)
%IF (result > "0")
  %{ perform some sort of error processing %}
%ELSE
%{ continue with normal processing %}
%ENDIF
```

Design your database and queries so that user requests do not have access to sensitive data about other users

Some database designs collect sensitive user data in a single table.

Unless SQL SELECT requests are qualified in some fashion, this approach may make all of the sensitive data available to any user at a web browser.

Example: The following SQL statement returns order information for an order identified by the variable `order_rn`:

```
select setsstatcode, setsfailtype, mestname
from   merchant, setstatus
where  merfnbr   = setsmenbr
and    setsornbr = $(order_rn)
```

This method permits users at a browser to specify random order numbers and possibly obtain sensitive information about the orders of other customers. One way to safeguard against this type of exposure is to make the following changes:

- Add a column to the order information table that identifies the customer associated with the order information within a specific row.
- Modify the SQL SELECT statement to ensure that the SELECT is qualified by an authenticated customer ID provided by the user at the browser.

For example, if `shlogid` is the column containing the customer ID associated with the order, and SESSION_ID is a Net.Data variable that contains the authenticated ID of the user at the browser, then you can replace the previous SELECT statement with the following statement:

```
select setsstatcode, setsfailtype, mestname
  from   merchant, setstatus
  where  merfnbr   = setsmenbr
  and    setsornbr = $(order_rn)
  and    shlogid   = $(SESSION_ID)
```

Use Net.Data hidden variables

You can use Net.Data hidden variables to conceal various characteristics of your Net.Data macro from users that view your HTML source with their Web browser. For example, you can hide the internal structure of your database. See “Hidden Variables” on page 69 for more information about hidden variables.

Request validation information from a user

You can create your own protection scheme based on user-provided input. For example, you can request validation information from a user through an HTML form and validate it using data that your Net.Data macro retrieves from a database or by calling an external program from a function defined in your Net.Data macro.

For more information on protecting your assets, see the Internet security list of frequently asked questions (FAQ) at this Web site:

http://www.w3.org/Security/Faq

Chapter 4. Invoking Net.Data

Net.Data for OS/400 is invoked using Common Gateway Interface (CGI) and using a macro. This type of invocation method is called macro request. Additionally, you can invoke persistent macros, or macros that contain functions specifying transaction boundaries. For more information about persistent macros, see "Chapter 7. Transaction Management with Persistent Macros" on page 133

This chapter describes invoking Net.Data with a macro.

- "Invoking Net.Data with a Macro (Macro Request)"
- "Invoking a Persistent Macro" on page 48

Invoking Net.Data with a Macro (Macro Request)

A client browser invokes Net.Data by sending a request in the form of a URL. This section shows you how to invoke Net.Data by specifying a macro in the URL request.

The request sent to Net.Data has the following form.

`http://`*`server`*`/`*`Net.Data_invocation_path`*`/`*`filename`*`/`*`block`*`[?`*`name`*`=`*`val`*`&...]`

Parameters:

server Specifies the name and path of the Web server. If the server is the local server, you can omit the server name and use a relative URL.

Net.Data_invocation_path
The path and filename of the Net.Data executable file. For example, `/cgi-bin/db2www/`.

filename
Specifies the name of the Net.Data macro file. Net.Data searches for and tries to match this file name with the path statements defined in the MACRO_PATH initialization path variable. See "MACRO_PATH" on page 23 for more information.

block Specifies the name of the HTML block in the referenced Net.Data macro.

?*name***=***val***&...**
Specifies one or more optional parameters passed to Net.Data.

You specify this URL directly in your browser. You can also specify it in an HTML link or build it using a form as follows:

© Copyright IBM Corp. 1997, 2001

- HTML link:

  ```
  <a href="URL">any text</a>
  ```

- HTML form:

  ```
  <form method="method" ACTION="URL">any text</form>
  ```

Parameters:

method Specifies the HTML method used with the form.

URL Specifies the URL used to run the Net.Data macro, the parameters of which are described above.

Examples

The following examples demonstrate the different methods of invoking Net.Data.

Example 1: Invoking Net.Data using an HTML link:

```
<a href="http://server/cgi-bin/db2www/myMacro.dtw/report">
.
.
.
</a>
```

Example 2: Invoking Net.Data using a form

```
<form method="post"
 action="http://server/cgi-bin/db2www/myMacro.dtw/report">
.
.
.
</form>
```

Example 3: Invoking Net.Data macros in the qsys.lib file system, using an HTML link:

```
<a href="http://server/cgi-bin/db2www/myMacro.mbr/report">
.
.
.
</a>
```

Example 4: Invoking Net.Data macros in the qsys.lib file system, using a form:

```
<form method=post
  action="http://server/cgi-bin/db2www/
  qsys.lib/mylib.lib/myfile.file/myMacro.mbr/report">
.
```

```
.
.
</form>
```

The following sections describe HTML links and forms and more about how to invoke Net.Data with them:

- "HTML Links"
- "HTML Forms" on page 46

HTML Links

If you are authoring a Web page, you can create an HTML link that results in the execution of an HTML block. When a user at a browser clicks on a text or image that is defined as an HTML link, Net.Data executes the HTML block within the macro.

To create an HTML link, use the HTML <a> tag. Decide which text or graphic you want to use as your hyperlink to the Net.Data macro, then surround it by the <a> and tags. In the HREF attribute of the <a> tag, specify the macro and the HTML block.

The following example shows a link that results in the execution of an SQL query when a user selects the text ″List all monitors″ on a Web page.

```
<a href="http://server/netdata-cgi/db2www/listA.d2w/report?hdware=mon">
List all monitors</a>
```

Clicking on the link calls a macro named listA.dtw, which has an HTML block named ″report″, as in the following example:

```
%DEFINE DATABASE="MNS97"

 %FUNCTION(DTW_SQL) myQuery(){
SELECT MODNO, COST, DESCRIP FROM EQPTABLE
WHERE TYPE='$(hdware)'
%REPORT{
<h3>Here is the list you requested</h3>
%ROW{
<hr />
$(N1): $(V1), $(N2): $(V2)
<p>$(N3): $(V3)</p>
%}
%}
%}

%HTML (Report){
@myQuery()
%}
```

The query returns a table that contains model number, cost, and description information for each monitor that is described within the EQPTABLE table.

The value of hdware in the SQL statement is taken from the URL input. See *Net.Data Reference* for a detailed description of the variables that are used in the ROW block.

HTML Forms

You can dynamically customize the execution of your Net.Data macros using HTML forms. Forms allow users to provide input values that can affect the execution of the macro and the contents of the Web page that Net.Data builds.

The following example builds on the monitor list example in "HTML Links" on page 45 by letting users at a browser use a simple HTML form to select the type of product for which information will be displayed.

```
<h1>Hardware Query Form</h1>
<hr>
<form method="post" action="/cgi-bin/db2www/listA.dtw/report">
<p>What type of hardware do you want to see?</p>
<ul>
<li><input type="radio" name="hdware" value="mon" checked /> Monitors</li>
<li><input type="radio" name="hdware" value="pnt" /> Pointing devices</li>
<li><input type="radio" name="hdware" value="prt" /> Printers</li>
<li><input type="radio" name="hdware" value="scn" /> Scanners</li>
</ul>

<input type="submit" value="submit" />
</form>
```

After the user at the browser makes a selection and clicks on the Submit button, the Web server processes the ACTION parameter of the FORM tag, which invokes Net.Data. Net.Data then executes the macro listA.dtw, which has an HTML block named ″report″ as listed above.

```
%DEFINE DATABASE="MNS97"

 %FUNCTION(DTW_SQL) myQuery(){
SELECT MODNO, COST, DESCRIP FROM EQPTABLE
WHERE TYPE='$(hdware)'
%REPORT{
<h3>Here is the list you requested</h3>
%ROW{
<hr />
$(N1): $(V1), $(N2): $(V2)
<p>$(N3): $(V3)</p>
%}
%}
%}

%HTML (Report){
@myQuery()
%}
```

In the above example, the value of `hdware` in the SQL statement is taken from the HTML form input. See *Net.Data Reference* for a detailed description of the variables that are used in the ROW block.

An input type that is given special treatment by Net.Data is the FILE input type. With this input type, users can upload a file to the server, which can be further processed by Net.Data or any other application on the server.

Net.Data tags the files with the correct codepage. The uploaded files are stored in the directory specified in DTW_UPLOAD_DIR and are given a unique name, determined using the following rules:

Syntax:

MacroFileName + '.' + *FormVarName* + '.' + *UniqueIdentifier* + '.' + *FormFileName*

MacroFileName
The name of the macro handling the request (the one called in the form). Only the filename is used, not the complete path.

FormVarName
The name of the variable used to identify the file in the form.

UniqueIdentifier
A string used to ensure uniqueness.

Example:

First, set DTW_UPLOAD_DIR in the Net.Data initialization file:

```
DTW_UPLOAD_DIR /tmp/uploads
```

Then, construct a form that invokes a macro and uses at least one input tag of type *file*.

```
<form method="post" enctype="multipart/form-data"
      action="/netdatadev/form.dtw/report">
Name:  <input type="text" name="name" /><br />
Zip code:  <input type="text" name="zipno" /><br />
Resume:  <input type="file" name="resume" /><br />
  <input type="submit" />
</form>
```

If a user were to submit the form, specifying the file `myresume.txt`, the resulting file would be written on the server with a name similar to:

```
/tmp/uploads/form.dtw.resume.20010108112341275-6245-021.myresume.txt
```

Invoking a Persistent Macro

This section shows you how to invoke persistent macros. These macros contain functions used for transaction processing. Invoking these macros is similar to regular macro request, in which you specify a server, macro, and HTML block. For persistent macros, you also specify a transaction handle, which identifies the HTML block as part of a transaction.

For more information about persistent macros and transaction processing, see "Chapter 7. Transaction Management with Persistent Macros" on page 133.

Persistent Macro Syntax

Use the following syntax to invoke a persistent macro:

- HTML link:

  ```
  <a href="http://server/Net.Data_invocation_path/transaction_handle/filename/
  block/[?name=val&...]">any text</a>
  ```

- HTML form:

  ```
  <form method=method ACTION="http://server/Net.Data_invocation_path/
  transaction_handle/filename/block/
  [?name=val&...]">any text</form>
  ```

- URL:

  ```
  http://server/Net.Data_invocation_path/transaction_handle/filename/block/
  [?name=val&...]
  ```

Parameters:

server Specifies the name of the Web server. If the server is the local server, you can omit the server name and use a relative URL.

Net.Data_invocation_path
The path and filename of the Net.Data executable file. For example, `/cgi-bin/db2www/`.

transaction_handle
Specifies which URLs are part of a transaction initiated by a Net.Data macro. This identifier is obtained by calling the DTW_RTVHANDLE built-in function and must follow the *Net.Data_invocation_path.*

filename
Specifies the name of the Net.Data macro. Net.Data searches for and tries to match this file name with the path statements defined in the MACRO_PATH initialization path variable. See "MACRO_PATH" on page 23 for more information.

block Specifies the name of the HTML block in the referenced Net.Data macro.

method Specifies the HTML method used with the form.

?*name***=***val***&...**

Specifies one or more optional parameters passed to Net.Data.

Examples

The following examples demonstrate how to invoke persistent macros.

Example 1: A URL in a macro:

```
http://www.mycompany.com/cgi-bin/db2www/$(handle)/mymacro.mac/report1
```

Example 2: A typical HTML block with links to other macro invocations that run in the same transaction

```
@DTW_STATIC()
 ...
%define handle = ""
@DTW_RTVHANDLE(handle)

%html(report) {
@DTW_ACCEPT(handle)
 ...
<a href="/cgi-bin/db2www/$(handle)/qsys.lib/mylib.lib/macros.file/
  pcgi1.mbr/report2">continue</a><br />
<a href="/cgi-bin/db2www/$(handle)/qsys.lib/mylib.lib/macros.file/
  pcgi1.mbr/quit">quit</a><br />
%}
```

Chapter 5. Developing Net.Data Macros

A Net.Data macro is a text file consisting of a series of Net.Data macro language constructs that:

- Specify the layout of Web pages
- Define variables and functions
- Call functions that are built-in to Net.Data or defined in the macro
- Format the processing output and return it to the Web browser for display

The Net.Data macro contains two organizational parts: the declaration part and the presentation part, as shown in Figure 6.

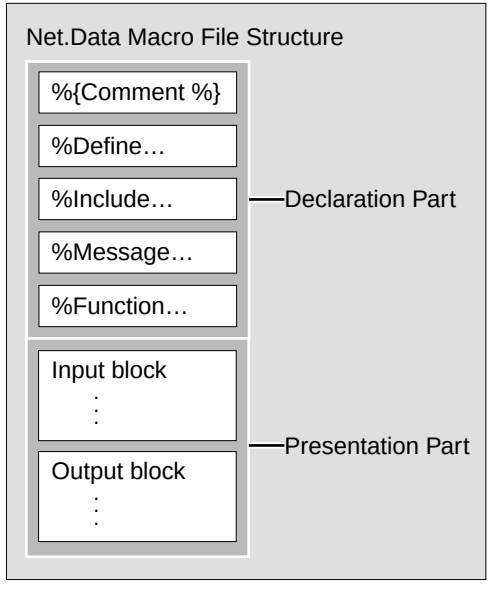

Figure 6. Macro Structure

- The *declaration part* contains the definitions of variables and functions in the macro.
- The *presentation part* contains HTML or XML blocks that specify the layout of the Web page. The HTML or XML blocks are made up of text presentation statements that are supported by your Web browser, such as HTML, JavaScript, and well-formed XML.

You can use these parts multiple times and in any order. See *Net.Data Reference* for syntax of the macro parts and constructs.

© Copyright IBM Corp. 1997, 2001

Authorization Tip: Ensure that the user ID under which Net.Data executes has the authorization to read this file. See "Granting Access Rights to Objects Accessed by Net.Data" on page 29 for more information.

This chapter examines the different blocks that make up a Net.Data macro and methods you can use for writing the macro.

Anatomy of a Net.Data Macro

The macro consists of two parts:

- The declaration part, that contains definitions used in the presentation part. The declaration part uses two major optional blocks:
  - DEFINE block
  - FUNCTION block

  The declaration part can also contain other language constructs and statements, such as EXEC statements, IF blocks, INCLUDE statements, and MESSAGE blocks. For more information about the language constructs, see the chapter about language constructs in *Net.Data Reference*.

  Authorization Tip: Ensure that the user ID under which Net.Data executes has authorization to read and execute files referenced by EXEC statements and to read files referenced by INCLUDE statements. See "Granting Access Rights to Objects Accessed by Net.Data" on page 29 for more information.
- The presentation part defines the layout of the Web page, references variables, and calls functions using HTML or XML blocks that are used as entry and exit points from the macro. When you invoke Net.Data, you specify a block name as an entry point for processing the macro. The HTML or XML blocks are described in "HTML Blocks" on page 55 and "XML Blocks" on page 57.

In this section, a simple Net.Data macro illustrates the elements of the macro language. This example macro presents a form that prompts for information to pass to a REXX program. The macro passes this information to an external REXX program called `ompsamp.mbr`, which echoes the data that the user enters. The results are then displayed on a second Web page.

First, look at the entire macro, and then each block in detail:

```
%{ **********************       DEFINE block        ************************%}
%DEFINE {
   page_title="Net.Data Macro Template"
%}

%{ ********************** FUNCTION Definition block ************************%}
%FUNCTION(DTW_REXX) rexx1 (IN input) returns(result)
{
  %EXEC{ompsamp.mbr %}
%}

%FUNCTION(DTW_REXX) today () RETURNS(result)
{
      result = date()
%}

%{ **********************      HTML Block: Input    ************************%}
%HTML (INPUT) {
<html>
<head>
<title>$(page_title)</title>
</head><body>
<h1>Input Form</h1>
Today is @today()

<form method="post" action="output">
Type some data to pass to a REXX program:
<input name="input_data" type="text" size="30" />
<p>
<input type="submit" value="enter" />
</p>
</form>

<hr>
<p>[<a href="/">Home page</a>]
</body></html>
%}

%{ **********************    HTML Block: Output     ************************%}
%HTML (OUTPUT) {
<html>
<head>
<title>$(page_title)</title>
</head><body>
<h1>Output Page</h1>
<p>@rexx1(input_data)
<p><hr>
<p>[<a href="/">Home page</a> |
<a href="input">Previous page</a>]
</body></html>
%}
```

The sample macro consists of four major blocks: the DEFINE, the FUNCTION, and the two HTML blocks. You can have multiple DEFINE, FUNCTION, and HTML blocks in one Net.Data macro.

The two HTML blocks contain text presentation statements such as HTML, which make writing Web macros easy. If you are familiar with HTML, building a macro simply involves adding macro statements to be processed dynamically at the server and SQL statements to send to the database.

Although the macro looks similar to an HTML document, the Web server accesses it through Net.Data using CGI. To invoke a macro, Net.Data requires two parameters: the name of the macro to process, and the HTML block in that macro to display.

When the macro is invoked, Net.Data processes it from the beginning. The following sections look at what happens as Net.Data processes the file.

The DEFINE Block

The DEFINE block contains the DEFINE language construct and variable definitions used later in the HTML blocks. The following example shows a DEFINE block with one variable definition:

```
%{ ********************       DEFINE Block        ************************%}
%DEFINE {
   page_title="Net.Data Macro Template"
%}
```

The first line is a comment. A comment is any text inside %{ and %}. Comments can be anywhere in the macro. The next statement starts a DEFINE block. You can define multiple variables in one define block. In this example, only one variable, page_title, is defined. After it is defined, this variable can be referenced anywhere in the macro using the syntax, $(page_title). Using variables makes it easy to make global changes to your macro later. The last line of this block, %}, identifies the end of the DEFINE block.

The FUNCTION Block

The FUNCTION block contains declarations for functions invoked by the HTML blocks. Functions are processed by language environments and can execute programs, SQL queries, or stored procedures.

The following example shows two FUNCTION blocks. One defines a call to an external REXX program and the other contains inline REXX statements.

```
%{ ********************* FUNCTION Block ***********************************%}
%FUNCTION(DTW_REXX) rexx1 (IN input) returns(result) { <-- This function accepts
                                                         one parameter and returns the
                                                         variable 'result', which is
                                                         assigned by the external REXX
                                                         program
    %EXEC{ompsamp.mbr %}  <-- The function executes an external REXX program
```

```
                                called "ompsamp.mbr"
%}

%FUNCTION(DTW_REXX) today () RETURNS(result) {
      result = date()  <-- The single source statement for this function is
                           contained inline.
%}
```

The first function block, rexx1, is a REXX function declaration that in turn, runs an external REXX program called ompsamp.mbr. One input variable, input, is accepted by this function and automatically passed to the external REXX command. The REXX command also returns one variable called result. The contents of the result variable in the REXX command replaces the invoking @rexx1() function call contained in the OUTPUT block. The variables input and result are directly accessible by the REXX program, as shown in the source code for ompsamp.mbr:

```
/* REXX */
result = 'The REXX program received "'input'" from the macro.'
```

The code in this function echoes the data that was passed to it. You can format the resulting text any way you want by enclosing the requesting @rexx1() function call in normal mark-up style tags (like <b> or). Rather than using the result variable, the REXX program could have written HTML tags to standard output using REXX SAY statements.

The second function block, also refers to a REXX program, today. However, the entire REXX program in this case is contained in the function declaration itself. An external program is not needed. Inline programs are allowed for both REXX and Perl functions because they are interpreted languages that can be parsed and executed dynamically. Inline programs have the advantage of simplicity by not requiring a separate program file to manage. The first REXX function could also have been handled inline.

HTML Blocks

HTML blocks define the layout of the Web page, reference variables, and call functions. HTML blocks are used as entry and exit points from the macro. An HTML block is always specified in the Net.Data macro request and every macro must have at least one HTML block.

The first HTML block in the example macro is named INPUT. The HTML(INPUT) contains the HTML for a simple form with one input field.

```
%{ **********************      HTML Block: Input    ************************%}
%HTML (INPUT) {               <--- Identifies the name of this HTML block.
<html>
<head>
<title>$(page_title)</title>  <--- Note the variable substitution.
</head><body>
<h1>Input Form</h1>
```

```
Today is @today()             <--- This line contains a call to a function.

<form method="post" action="output">  <--- When this form is submitted,
                                           the "OUTPUT" HTML block is called.<p>
Type some data to pass to a REXX program:
<input name="input_data"      <--- "input_data" is defined when the form
TYPE="text" SIZE="30" />           is submitted and can be referenced elsewhere in
                                   this macro. It is initialized to whatever the
                                   user types into the input field.
</p>
<input type="submit" value="enter" />

<hr>
<p>
[
<a href="/">Home page</a>]</p>
 </body><html>
%}                            <--- Closes the HTML block.
```

The entire block is surrounded by the HTML block identifier, `%HTML (INPUT) {...%}`. `INPUT` identifies the name of this block. The name can contain underscores, periods, and any alphanumeric character; Net.Data does not distinguish by case. The HTML `<title>` tag contains an example of variable substitution. The value of the variable `page_title` is substituted into the title of the form.

This block also has a function call. The expression `@today()` is a call to the function `today`. This function is defined in the FUNCTION block that is described above. Net.Data inserts the result of the `today` function, the current date, into the HTML text in the same location that the `@today()` expression is located.

The ACTION parameter of the FORM statement provides an example of navigation between HTML blocks or between macros. Referencing the name of another block in an ACTION parameter accesses that block when the form is submitted. Any input data from an HTML form is passed to the block as implicit variables. This is true of the single input field defined on this form. When the form is submitted, data entered in this form is passed to the `HTML(OUTPUT)` block in the variable *input_data*.

You can access HTML blocks in other macros with a relative reference if the macros are on the same Web server. For example, the ACTION parameter `ACTION="../othermacro.dtw/main"` accesses the HTML block called main in the macro `othermacro.dtw`. Again, any data entered into the form is passed to this macro in the variable *input_data*.

When you invoke Net.Data, you pass the variable as part of the URL. For example:

```
<a href="/cgi-bin/db2www/othermacro.dtw/main?input_data=value">Next macro</a>
```

You can access or manipulate form data in the macro by referencing the variable name specified in the form.

The next HTML block in the example is the HTML(OUTPUT) block. It contains the HTML tagging and Net.Data macro statements that define the output processed from the HTML(INPUT) request.

```
%{ **********************    HTML Block: Output     ************************%}
%HTML (OUTPUT) {
<html>
<head>
<title>$(page_title)</title>  <--- More substitution.

</head><body>
<h1>Output Page</h1>
<p>@rexx1(input_data)  <--- This line contains a call to function rexx1
                           passing the argument "input_data".
<p>
<hr>
<p>
[
<a href="/">Home page</a> |
<a href="input">Previous page</a>]
%}
```

Like the HTML(INPUT) block, this block is standard HTML with Net.Data macro statements to substitute variables and a function call. Again the page_title variable is substituted into the title statement. And, as before, this block contains a function call. In this case, it calls the function rexx1 and passes to it the contents of the variable input_data, which it received from the form defined in the Input block. You can pass any number of variables to and from a function. The function definition specifies the number and the usage of variables that are passed.

XML Blocks

Whether you want to deliver XML to another processing application or to a client browser, you can use the XML block structure to deliver XML content.

The XML block works in the same manner as the HTML block; it is an entry point to the macro. Within the block you can enter XML tags directly, reference variables, and make function calls.

So that you can customize the generated XML document to your needs, the XML block does not generate the prolog tags. Enter the prolog information particular to your enterprise and include a stylesheet of your choice. Included with Net.Data are three XSL stylesheets that you can use. These stylesheets contain transforms for all of the XML elements generated by Net.Data. The stylesheets are examples, however, and you are encouraged to expand on these or create your own.

```
%DEFINE SHOWSQL = "yes"

%FUNCTION(DTW_SQL) NewManager(){
select * from staff where job = 'Mgr' and years <= 5
%}

%XML(report) {
<?xml version="1.0" ?>
<?xml-stylesheet type="text/xsl" href="/netdata-xml/ndTable.xsl" ?>

<XMLBlock>
  <h1>List of New Managers</h1>
  @NewManager()
</XMLBlock>
%}
```

Figure 7. A macro containing an XML report block

When calling an SQL function that returns a default report, Net.Data generates the result set using a small set of XML elements, as shown in the following sample Document Type Description (DTD).

```
<!-------------------------------------------------------->
<!-- The root element of the document.                  -->
<!-------------------------------------------------------->
<!ELEMENT XMLBlock (RowSet|ShowSQL|Message)*>
<!ATTLIST XMLBlock name   CDATA #IMPLIED>

<!-------------------------------------------------------->
<!-- The default presentation format for tables uses    -->
<!-- the RowSet, Row, and Column elements.              -->
<!-------------------------------------------------------->
<!ELEMENT RowSet   (Row)*>
<!ATTLIST RowSet   name   CDATA #IMPLIED>
<!ELEMENT Row      (Column)*>
<!ATTLIST Row      name   CDATA #IMPLIED
                   number CDATA #IMPLIED>
<!ELEMENT Column   (#PCDATA)>

<!-------------------------------------------------------->
<!-- SQL statements resulting from setting the SHOWSQL  -->
<!-- variable are presented with the ShowSQL element.   -->
<!-------------------------------------------------------->
<!ELEMENT ShowSQL  (#PCDATA)>

<!-------------------------------------------------------->
<!-- Messages are presented with the Message element.   -->
<!-------------------------------------------------------->
<!ELEMENT Message  (#PCDATA)>
```

The elements are defined as follows:

XMLBlock
The root element for the document. This tag must be entered manually.

RowSet
Contains the rows in a result set. The name attribute of RowSet is determined as follows:

- For a result set returned from a call to a function that executes an SQL query, the name of the function is used.
- For a result set returned from a call to a stored procedure, the name of the result set is used. If the result set is not named, then the function name is used.

Row Contains the columns of a row and is numbered for identification.

Column
Contains the data value for the particular row and the column by which it is named.

ShowSQL
Contains the SQL statement for the current query.

Message
Contains any error message produced by Net.Dta or DB2.

Using the elements above, Net.Data would generate the following output from the macro listed in Figure 7 on page 58.

```
<?xml version="1.0" ?>
<?xml-stylesheet type="text/xsl" href="/netdata-xml/ndTable.xsl" ?>
<XMLBlock>
  <h1>List of New Managers</h1>
  <ShowSQL>select * from staff where job = 'Mgr' and years <= 5</ShowSQL>
  <RowSet name="NewManager">
    <Row number="1">
      <Column name="ID">30</Column>
      <Column name="NAME">Marenghi</Column>
      <Column name="DEPT">38</Column>
      <Column name="JOB">Mgr</Column>
      <Column name="YEARS">5</Column>
      <Column name="SALARY">17506.75</Column>
      <Column name="COMM"></Column>
    </Row>
    <Row number="2">
      <Column name="ID">240</Column>
      <Column name="NAME">Daniels</Column>
      <Column name="DEPT">10</Column>
      <Column name="JOB">Mgr</Column>
      <Column name="YEARS">5</Column>
      <Column name="SALARY">19260.25</Column>
```

```
      <Column name="COMM"></Column>
    </Row>
  </RowSet>
</XMLBlock>
```

Figure 8 and Figure 9 on page 61 show how the above data would appear in a browser using each of the two stylesheets provided with Net.Data: ndTable.xsl and ndRecord.xsl.

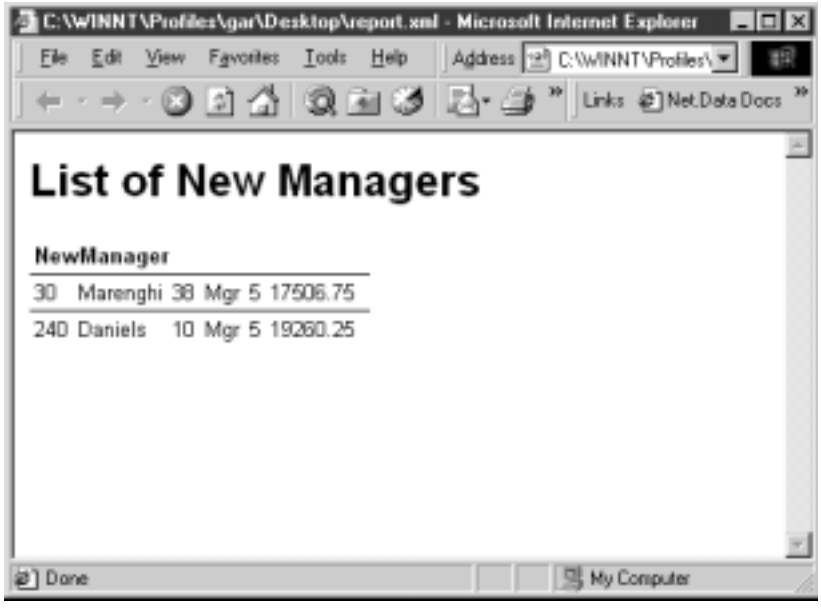

Figure 8. XML displayed using the ndTable.xsl stylesheet

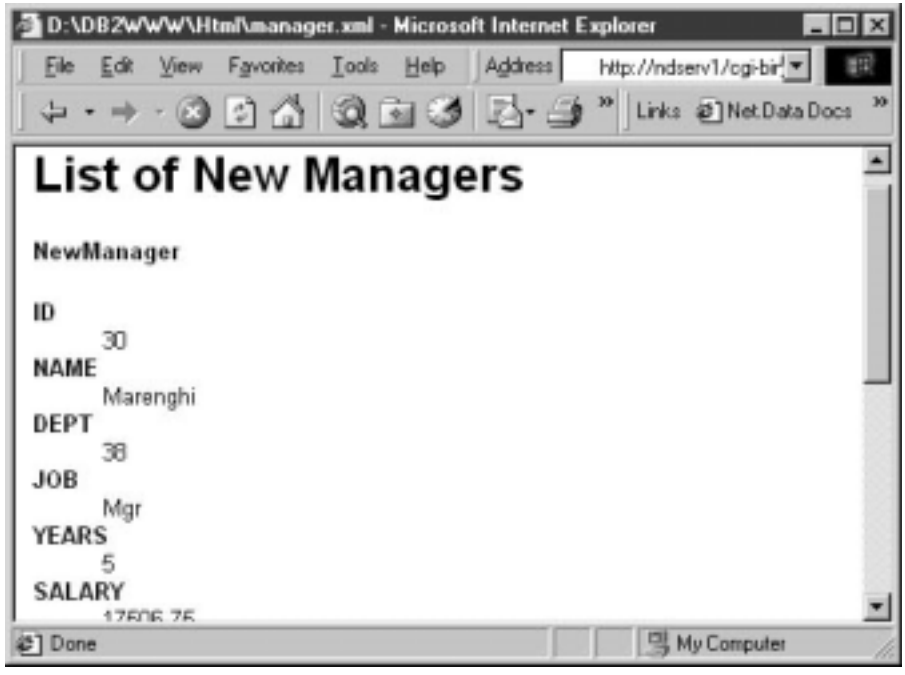

Figure 9. XML displayed using the ndRecord.xsl stylesheet

Net.Data Macro Variables

Net.Data lets you define and reference variables in a Net.Data macro. In addition, you can pass these variables from the macro to the language environments and back. The variable names, values, and literal strings that are passed are called tokens. Net.Data puts no limit on the size of the tokens and will pass any token that the memory of your system can handle. Individual language environments, however, might provide restrictions on the token size.

Net.Data variables can be defined depending on the type of variable and whether it has a predefined value. These variables can be categorized into the following types, based on how they are defined:

- Explicitly defined variables using the DEFINE statement in the DEFINE block
- Predefined variables, which are variables that are made available by Net.Data and are set to a value. This value usually cannot be changed.
- Implicitly defined variables, which are of four types:
  - Variables that are not explicitly defined but are instantiated when first assigned a value.

– Parameter variables that are part of a FUNCTION block definition and that can only be referenced within a FUNCTION block.
– Variables that are instantiated by Net.Data and correspond to form data or query string data.
– Variables that are associated with a Net.Data table and that can only be referenced within a ROW block or REPORT block.

The following sections describe:

Identifier Scope

If an identifier has global scope, then it can be referenced anywhere in a macro during a single request. The region where an identifier is visible is called its *scope*. The five types of scope are:

• Global

  An identifier has global scope if you can reference it anywhere within a macro. Identifiers that have global scope are:
  – Net.Data built-in functions
  – Form data
  – Query string data
  – Variables instantiated from within an HTML block

• Macro

  An identifier has this scope if its declaration appears outside of any block. A block starts with an opening bracket ({) and ends with a percent sign and bracket (%}). (DEFINE blocks are excluded from this definition.) Unlike an identifier with global scope, one with macro scope can only be referred to by items in the macro that follow the idenfier's declaration.

• FUNCTION block or MACRO_FUNCTION block

  An identifier has function block scope if:

  – The identifier is declared in the parameter list of the function definition.

    If an identifier with the same name already exists outside the function definition, then Net.Data uses the identifier from the function parameter list within the function block.

  – The identifier is instantiated in the function block and is not declared or instantiated prior to the function call.

  An identifier does not have function block scope if it has been declared or initialized outside of the function and is not declared in the function

parameter list. The value of the identifier within the function block remains unchanged unless updated by the function.

- REPORT block

  An identifier has report block scope if it can be referenced only from within a REPORT block (for example, table column names N1, N2, ..., N*n*). Only those variables that Net.Data implicitly defines as part of its table processing can have a report block scope. Any other variables that are instantiated have function block scope.

- ROW block

  An identifier has row block scope if it can only be referenced from within a ROW block (for example, table value names V1, V2, ..., Vn). Only those variables that Net.Data implicitly defines as part of its table processing can have a row block scope. Any other variables that are instantiated have function block scope.

Defining Variables

There are three ways to define variables in a Net.Data macro:

- Define statement or block
- HTML form tags
- Query string data

A variable value received from form or query string data overrides a variable value set by a DEFINE statement in a Net.Data macro.

- **DEFINE statement or block**

  The simplest way to define a variable for use in a Net.Data macro is to use the DEFINE statement. The syntax is as follows:

```
%DEFINE variable_name="variable value"

%DEFINE variable_name={ variable value on multiple
                        lines of text  %}

%DEFINE {
    variable_name1="variable value 1"
    variable_name2="variable value 2"
%}
```

  The *variable_name* is the name you give the variable. Variable names must begin with a letter or underscore and can contain any alphanumeric character, an underscore, a period, or a hash (#). All variable names are case-sensitive, except V_*columnName*, which is a table variable.

  For example:

```
%DEFINE reply="hello"
```

  The variable `reply` has the value `hello`.

Two consecutive quotes alone is equal to an empty string. For example:

```
%DEFINE empty=""
```

The variable empty has an empty string.

If your variable contains special characters, such as an end-of-line, use block braces around the value:

```
%DEFINE introduction={
Hello,
My name is John.
%}
```

To include quotes in a string, you can use two quotes consecutively.

```
%DEFINE HI="say ""hello"""
```

You can also use block braces to escape the quotes:

```
%DEFINE HI={ say "hello" %}
```

To define several variables with one DEFINE statement, use a DEFINE block:

```
%DEFINE {
         variable1="value1"
         variable2="value2"
         variable3="value3"
         variable4="value4"
%}
```

- **HTML form tags: SELECT, INPUT, and TEXTAREA**

  You can use HTML FORM tags to assign values to variables, namely the SELECT, INPUT, and TEXTAREA tags. The following example uses standard HTML form tags to define Net.Data variables:

  ```
  <input name="variable_name" TYPE=... />
  ```

  or

  ```
  <select name="variable_name">
    <option>value one
    <option>value two
  </select>
  ```

  To assign a variable that spans multiple lines or contains special characters, such as quotes, the TEXTAREA tag can be used:

  ```
  <textarea name="variable_name" ROWS="4">
  Please type the multi-line value
  of your variable here.
  </textarea>
  ```

The *variable_name* is the name you give the variable, and the value of the variable is determined from the input received in the form. See "HTML Forms" on page 46 for an example of how this type of variable definition is used in a Net.Data macro.

- **Query string data**

  You can pass variables to Net.Data through the query string. For example:

  ```
  http://www.ibm.com/cgi-bin/db2www/stdqry1.dtw/input?field=custno
  ```

  In the above example, the variable name, `field`, and the variable value, `custno`, specify additional data that Net.Data receives from the query string. Net.Data receives and processes the data as it would from form data.

Referencing Variables

You can reference a previously defined variable to return its value. To reference a variable in Net.Data macros, specify the variable name inside `$(` and `)`. For example:

```
$(variableName)
$(homeURL)
```

When Net.Data finds a variable reference, it substitutes the variable reference with the value of the variable. Variable references can contain strings, variable references, and function calls.

You can dynamically generate variable names. With this technique, you can use loops to process variably-sized tables or input data for lists that are built at run time, when the number in the list cannot be determined in advanced. For example, you can generate lists of HTML form elements that are generated based on records returned from an SQL query.

To use variables as part of your text presentation statements, reference them in the HTML blocks of your macro.

Invalid variable references: Invalid variable references are resolved to the empty string. For example, if a variable reference contains invalid characters such as an exclamation point (!), the reference is resolved to the empty string.

Valid variable names must begin with an alphanumeric character or an underscore, and they can consist of alphanumeric characters, including a period, underscore, and hash mark.

Example 1: Variable reference in a link

If you have defined the variable *homeURL*:

```
%DEFINE homeURL="http://www.ibm.com/"
```

You can refer to the home page as $(*homeURL*) and create a link:

```
<a href="$(homeURL)">Home page</a>
```

You can reference variables in many parts of the Net.Data macro; check the language constructs in this chapter to determine in which parts of the macro variable references are allowed. If the variable has not yet been defined at the time it is referenced, Net.Data returns an empty string. A variable reference alone does not define the variable.

Example 2: Dynamically generated variable references

Assume that you run an SQL SELECT statement with any number of elements. You can create an HTML form with input fields using the following ROW blocks:

```
...
%ROW {
<input type=text name=@dtw_rconcat("I", ROW_NUM) size=10 maxlength=10 />
%}
...
```

Because you created INPUT fields, you would probably want to access the values that the user entered when the form is submitted to your macro for processing. You can code a loop to retrieve the values in a variable length list:

```
<pre>
...
@dtw_assign(rowIndex, "1")
%while (rowIndex <= rowCount) {
The value entered for row $(rowIndex) is: $(I$(rowIndex))
@dtw_add(rowIndex, "1", rowIndex) %}
...
</pre>
```

Net.Data first generates the variable name using the I$(rowIndex) reference. For example, the first variable name would be I1. Net.Data then uses that value and resolves to the value of the variable.

Example 3: A variable reference with nested variable references and a function call

```
%define my = "my"
%define u = "lower"
%define myLOWERvar = "hey"


$($(my)@dtw_ruppercase(u)var)
```

The variable reference returns the value of hey.

Variable Types

You can use the following types of variables in your macros.

- "Conditional Variables"
- "Environment Variables"
- "Executable Variables" on page 68
- "Hidden Variables" on page 69
- "List Variables" on page 70
- "Table Variables" on page 71
- "Miscellaneous Variables" on page 72
- "Table Processing Variables" on page 73
- "Report Variables" on page 73
- "Language Environment Variables" on page 74

If you assign strings to variables that are defined a certain way by Net.Data, such as ENVVAR, LIST, condition list variables, the variable no longer behaves in the defined way. In other words, the variable becomes a simple variable, containing a string.

See *Net.Data Reference* for syntax and examples of each type of variable.

Conditional Variables

Conditional variables let you define a conditional value for a variable by using a method similar to an IF, THEN construct. When defining the conditional variable, you can specify two possible variable values. If the first variable you reference exists, the conditional variable gets the first value; otherwise the conditional variable gets the second value. The syntax for a conditional variable is:

```
varA = varB ? "value_1" : "value_2"
```

If varB is defined, varA="*value_1*", otherwise varA="*value_2*". This is equivalent to using an IF block, as in the following example:

```
%IF (varB)
    varA = "value_1"
%ELSE
    varA = "value_2"
%ENDIF
```

See "List Variables" on page 70 for an example of using conditional variables with list variables.

Environment Variables

You can reference environment variables that the Web server makes available to the process or thread that is processing your Net.Data request. When the

ENVVAR variable is referenced, Net.Data returns the current value of the environment variable by the same name.

The syntax for defining environment variables is:

```
%DEFINE var=%ENVVAR
```

Where *var* is the name of the environment variable being defined.

For example, the variable SERVER_NAME can be defined as environment variable:

```
%DEFINE SERVER_NAME=%ENVVAR
```

And then referenced:

```
The server is $(SERVER_NAME)
```

The output looks like this:

```
The server is www.ibm.com
```

See *Net.Data Reference* for more information about the ENVVAR statement.

Executable Variables

You can invoke other programs from a variable reference using executable variables.

Define executable variables in a Net.Data macro using the EXEC language construct in the DEFINE block. For more information about the EXEC language element, see the language constructs chapter in the *Net.Data Reference*. In the following example, the variable `runit` is defined to execute the executable program `testProg`:

```
%DEFINE runit=%EXEC "testProg"
```

`runit` becomes an executable variable.

Net.Data runs the executable program when it encounters a valid variable reference in a Net.Data macro. For example, the program `testProg` is executed when a valid variable reference is made to the variable `runit` in a Net.Data macro.

A simple method is to reference an executable variable from another variable definition. The following example demonstrates this method. The variable `date` is defined as an executable variable and `dateRpt` contains a reference to the executable variable.

```
%DEFINE date=%EXEC "date"
```

Wherever $(date) appears in the Net.Data macro, Net.Data searches for the executable program date, and when it locates it, displays:

```
Today is Tue 11-07-1999
```

When Net.Data encounters an executable variable in a macro, it looks for the referenced executable program using the following method:

1. It searches the directories specified by the EXEC_PATH in the Net.Data initialization file. See "EXEC_PATH" on page 21 for details.
2. If Net.Data does not locate the program, the system searches the directories defined by the system PATH environment variable or the library list. If it locates the executable program, Net.Data runs the program.

Restriction: Do not set an executable variable to the value of the output of the executable program it calls. In the previous example, the value of the variable date is NULL. If you use this variable in a DTW_ASSIGN function call to assign its value to another variable, the value of the new variable after the assignment is NULL also. The only purpose of an executable variable is to invoke the program it defines.

You can also pass parameters to the program to be executed by specifying them with the program name on the variable definition. In this example, the values of distance and time are passed to the program *calcMPH*.

```
%DEFINE mph=%EXEC "calcMPH $(distance) $(time)"
```

Hidden Variables

You can use hidden variables to conceal the actual name of a variable from application users who view your Web page source with their Web browser. To define a hidden variable:

1. Define a variable for each string you want to hide, after the variable's last reference in the HTML block. Variables are always defined with the DEFINE language construct after they are used in the HTML block, as in the following example. The $$(*variable*) variables are referenced and then defined.
2. In the HTML block where the variables are referenced, use double dollar signs instead of a single dollar sign to reference the variables. For example, $$(X) instead of $(X).

```
%HTML(INPUT) {
<form ...>
<p>Select fields to view:
shanghai<select name="field">
<option value="$$(name)"> Name
<option value="$$(addr)"> Address
...
</form>
%}
```

```
%DEFINE {
name="customer.name"
addr="customer.address"
%}

%FUNCTION(DTW_SQL) mySelect() {
  SELECT $(Field) FROM customer
%}

...
```

When a Web browser displays the HTML form, $$(name) and $$(addr) are replaced with $(name) and $(addr) respectively, so the actual table and column names never appear on the HTML form. Application users cannot tell that the true variable names are hidden. When the user submits the form, the HTML(REPORT) block is called. When @mySelect() calls the FUNCTION block, $(Field) is substituted in the SQL statement with customer.name or customer.addr in the SQL query.

List Variables

Use list variables to build a delimited string of values. They are particularly useful in helping you construct an SQL query with multiple items like those found in some WHERE or HAVING clauses. The syntax for a list variable is:

```
%LIST " value_separator " variable_name
```

Recommendation: The blanks are significant. Insert a space before and after the value separator for most cases. Most queries use Boolean or mathematical operators (for example, AND, OR, or >) for the value separator. The following example illustrates the use of conditional, hidden, and list variables:

```
%HTML(INPUT) {
<form method="post" action="/cgi-bin/db2www/example2.dtw/report">
<h2>Select one or more cities:</h2>
<input type="checkbox" name="conditions" value="$$(cond1)" />Sao Paolo<br />
<input type="checkbox" name="conditions" value="$$(cond2)" />Seattle<br />
<input type="checkbox" name="conditions" value="$$(cond3)" />Shanghai<br />
<input type="submit" value="submit query" />
</form>
%}

%DEFINE{
DATABASE="custcity"
 %LIST " OR " conditions
cond1="cond1='Sao Paolo'"
cond2="cond2='Seattle'"
cond3="cond3='Shanghai'"
whereClause= ? "WHERE $(conditions)"
%}

%FUNCTION(DTW_SQL) mySelect(){
SELECT name, city FROM citylist
```

```
$(whereClause)
%}

%HTML(REPORT){
@mySelect()
%}
```

In the HTML form, if no boxes are checked, `conditions` is empty, so `whereClause` is also empty in the query. Otherwise, `whereClause` has the selected values separated by OR. For example, if all three cities are selected, the SQL query is:

```
SELECT name, city FROM citylist
WHERE cond1='Sao Paolo' OR cond2='Seattle' OR cond3='Shanghai'
```

This example shows that `Seattle` is selected, which results in this SQL query:

```
SELECT name, city FROM citylist
WHERE cond1='Seattle'
```

Table Variables

The table variable defines a collection of related data. It contains a set of rows and columns including a row of column headers. A table is defined in the Net.Data macro as in the following statement:

```
%DEFINE myTable=%TABLE(30)
```

The number following `%TABLE` is the limit on the number of rows that this table variable can contain. To specify a table with no limit on the number of rows, use the default or specify ALL, as shown in these examples:

```
%DEFINE myTable2=%TABLE
%DEFINE myTable3=%TABLE(ALL)
```

When you define a table, it has zero rows and zero columns. The only way you can populate a table with values is by passing it as an OUT or INOUT parameter to a function or by using the built-in table functions provided by Net.Data. The DTW_SQL language environment automatically puts the results of a SELECT statement into a table.

For non-database language environments, such as DTW_REXX or DTW_PERL, the language environment is also responsible for setting table values. However, the language environment script or program defines the table values cell-by-cell. See “Chapter 6. Using Language Environments” on page 99 for more information about how language environments use table variables.

You can pass a table between functions by referring to the table variable name. The individual elements of the table can be referred to in a REPORT block of a function or by using the Net.Data table functions. See “Table Processing Variables” on page 73 for accessing individual elements in a table within a REPORT block, and see “Table Functions” on page 85 for accessing

individual elements of a table using a table function. Table variables are usually populated with values in an SQL function, and then used as input to a report, either in the SQL function or in another function after being passed to that function as a parameter. You can pass table variables as IN, OUT, or INOUT parameters to any non-SQL function. Tables can be passed to SQL functions only as OUT parameters.

If you reference a table variable, the contents of the table are displayed and formatted based on the setting of the DTW_HTML_TABLE variable. In the following example, the contents of `myTable` would be displayed:

```
%HTML (output) {
  $(myTable)
}
```

The column names and field values in a table are addressed as array elements with an origin of 1.

Miscellaneous Variables

These variables are Net.Data-defined variables that you can use to:

- Affect Net.Data processing
- Find out the status of a function call
- Obtain information about the result set of a database query
- Determine information about file locations and dates

Miscellaneous variables can either have a predefined value that Net.Data determines or have values that you set. For example, Net.Data determines the DTW_CURRENT_FILENAME variable value based on the current file that it is processing, whereas you can specify whether Net.Data removes extra white space caused by tabulators and new-line characters.

Predefined variables are used as variable references within the macro and provide information about the current status of files, dates, or the status of a function call. For example, to retrieve the name of the current file, you could use:

```
%REPORT {
  <p>This file is <i>$(DTW_CURRENT_FILENAME)</i>.</p>
}
```

Modifiable variable values are generally set using a DEFINE statement or with the @DTW_ASSIGN() function and let you affect how Net.Data processes the macro. For example, to specify whether white space is removed, you could use the following DEFINE statement:

```
%DEFINE DTW_REMOVE_WS="YES"
```

Table Processing Variables

Net.Data defines table processing variables for use in the REPORT and ROW blocks. Use these variables to reference values from SQL queries and function calls.

Table processing variables have a predefined value that Net.Data determines. These variables allow you to reference values from the result sets of SQL queries or function calls by the column, row, or field that is being processed. You can also access information about the number of rows being processed or a list of all the column names.

For example, as Net.Data processes a result set from an SQL query, it assigns the value of the variable Nn for each current column name, such that N1 is assigned to the first column, N2 is assigned to the second column, and so on. You can reference the current column name for your Web page output.

Use table processing variables as variable references within the macro. For example, to retrieve the name of the current column being processed, you could use:

```
%REPORT {
  <p>Column 1 is <i>$(N1)</i>.</p>
}
```

Table processing variables also provide information about the results of a query. You can reference the variable TOTAL_ROWS in the macro to show how many rows are returned from an SQL query, as in the following example:

```
Names found: $(TOTAL_ROWS)
```

Some of the table processing variables are affected by other variables or built-in functions. For example, TOTAL_ROWS requires that the DTW_SET_TOTAL_ROWS SQL language environment variable be activated so that Net.Data assigns the value of TOTAL_ROWS when processing the results from a SQL query or function call as in the following example:

```
%DEFINE DTW_SET_TOTAL_ROWS="YES"
...

Names found: $(TOTAL_ROWS)
```

Report Variables

Net.Data displays Web page output generated from the macro in a default report format. In an HTML block, the default report format displays a table using <pre> </pre> tags tags or using HTML table tags. In an XML block, <RowSet>, <Row>, and <Column> tags are used. You can override the default report by defining a REPORT block with instructions for displaying the output or by using one of the report variables to prevent the default report from being generated.

Report variables help you customize how your Web page output is displayed and how it is used with default reports and Net.Data tables. You must define these variables before using them with a DEFINE statement or with the @DTW_ASSIGN() function.

The report variables specify spacing, override default report formats, specify whether table output should be displayed in HTML or fixed-width characters, and specify other display features. For example, you can set DTW_HTML_TABLE to ″yes″ and Net.Data will generate the default report with HTML table tags instead of as a plain-text formatted table.

```
%DEFINE ALIGN="YES"
...
<p>Your query was on these columns: $(NLIST)
```

The START_ROW_NUM report variable lets you determine at which row to begin displaying the results of a query. For example, the following variable value specifies that Net.Data will begin displaying the results of a query at the third row.

```
%DEFINE START_ROW_NUM = "3"
```

You can also determine whether Net.Data uses HTML tags for default formatting. With DTW_HTML_TABLE set to YES, an HTML table is generated rather than a text-formatted table.

```
%DEFINE DTW_HTML_TABLE="YES"

%FUNCTION(DTW_SQL){
SELECT NAME, ADDRESS FROM $(qTable)
%}
```

Language Environment Variables

These variables are used with language environments and affect how the language environment processes a request.

With these variables, you can perform tasks such as establishing connections to databases, enabling NLS support, and determining whether the execution of an SQL statement is successful.

For example, you can use the SQL_STATE variable to access or display the SQL state value returned from the database.

```
%FUNCTION (DTW_SQL) val1() {
 select * from customer
%REPORT {
 ...
%ROW {
 ...
%}
 SQLSTATE=$(SQL_STATE)
%}
```

Net.Data Functions

Net.Data provides built-in functions for use in your applications, such as word and string manipulation functions or functions that retrieve and set table variable functions. You can also define functions for use with your application, for example to call an external program or a stored procedure.

User-defined functions
: Those functions that you define for use with your application, for example to call an external program or a stored procedure.

Net.Data built-in functions
: Those functions that Net.Data provides for use in your applications, such as functions for manipulating words and strings and functions that get and set table variables.

These sections describe the following topics:

- "Defining Functions"
- "Calling Functions" on page 80
- "Calling Net.Data Built-in Functions" on page 81

Defining Functions

To define your own functions in the macro, use a FUNCTION block or MACRO_FUNCTION block:

FUNCTION block
: Defines a subroutine that is invoked from a Net.Data macro and is processed by a language environment. FUNCTION blocks must contain language statements or calls to an external program.

MACRO_FUNCTION block
: Defines a subroutine that is invoked from a Net.Data macro and is processed by Net.Data rather than a language environment. MACRO_FUNCTION blocks can contain any statement that is allowed in an HTML block or XML block.

Syntax: Use the following syntax to define functions:

FUNCTION block:

```
%FUNCTION(type) function-name([usage] [datatype] parameter, ...)
                [RETURNS(return-var)] {
  executable-statements
  [report-block]
  ...
  [report-block]
  [message-block]
%}
```

MACRO_ FUNCTION block:

```
%MACRO_FUNCTION function-name([usage] parameter, ...) [RETURNS(return-var)] {
  executable-statements
  [report-block]
  ...
  [report-block]
%}
```

Where:

type Identifies a language environment that is configured in the initialization file. The language environment invokes a specific language processor (which processes the executable statements) and provides a standard interface between Net.Data and the language processor.

function-name
Specifies the name of the FUNCTION or MACRO_FUNCTION block. A function call specifies the *function-name*, preceded by an at (@) sign. See "Calling Functions" on page 80 for details.

You can define multiple FUNCTION or MACRO_FUNCTION blocks with the same name so that they are processed at the same time. Each of the blocks must all have identical parameter lists. When Net.Data calls the function, all FUNCTION blocks with the same name or MACRO_FUNCTION blocks with the same name are executed in the order they are defined in the Net.Data macro.

usage Specifies whether a parameter is an input (IN) parameter, an output (OUT) parameter, or both types (INOUT). This designation indicates whether the parameter is passed into or received back from the FUNCTION block, MACRO_FUNCTION block, or both. The usage type applies to all of the subsequent parameters in the parameter list until changed by another usage type. The default type is IN.

datatype
The data type of the parameter. Some language environments expect data types for the parameters that are passed. For example, the SQL language environment expects them when calling stored procedures, as does the Direct Call language environment when calling programs. See "Chapter 6. Using Language Environments" on page 99 to learn more about the supported data types for the language environment you are using.

parameter
The name of a variable with local scope that is replaced with the value of a corresponding argument specified on a function call. Parameters are passed to the language environment and are accessible to the executable statements using the natural syntax of that language or as environment variables. Parameter variable references are not valid outside the FUNCTION or MACRO_FUNCTION blocks.

return-var

Specify this parameter after the RETURNS keyword to identify a special OUT parameter. The value of the return variable is assigned in the function block, and its value is returned to the place in the macro from which the function was called. For example, in the following sentence, `<p>My name is @my_name().`, `@my_name()` gets replaced by the value of the return variable. If you do not specify the RETURNS clause, the value of the function call is:

- NULL if the return code from the call to the language environment is zero
- The value of the return code, when the return code is non-zero.

executable-statements

The set of language statements that is passed to the specified language environment for processing after the variables are substituted and the functions are processed. *executable-statements* can contain Net.Data variable references and Net.Data function calls.

For FUNCTION blocks, Net.Data replaces all variable references with the variable values, executes all function calls, and replaces the function calls with their resulting values before the executable statements are passed to the language environment. Each language environment processes the statements differently. For more information about specifying executable statements or calling executable programs, see “Executable Variables” on page 68.

For MACRO_FUNCTION blocks, the executable statements are a combination of text and Net.Data macro language constructs. In this case, no language environment is involved because Net.Data acts as the language processor and processes the executable statements.

report-block

Defines one or more REPORT blocks for handling the output of the FUNCTION or MACRO_FUNCTION block. See “Report Blocks” on page 88.

message-block

Defines the MESSAGE block, which handles any messages for error conditions returned by the FUNCTION block. For more information on how to capture error conditions, see “Message Blocks” on page 79.

Define functions outside of any other block and before they are called in the Net.Data macro.

Using Special Characters in Functions

When characters that match Net.Data language constructs syntax are used in the language statements section of a function block as part of syntactically

valid embedded program code (such as REXX or Perl), they can be misinterpreted as Net.Data language constructs, causing errors or unpredictable results in a macro.

For example, a Perl function might use the COMMENT block delimiter characters, %{. When the macro is run, the %{ characters are interpreted as the beginning of a COMMENT block. Net.Data then looks for the end of the COMMENT block, which it thinks it finds when it reads the end of the function block. Net.Data then proceeds to look for the end of the function block, and when it can't be found, issues an error.

Use one of the following methods to use COMMENT block delimiter characters, or any other Net.Data special characters as part of your embedded program code, without having them interpreted by Net.Data as special characters:

- Use the EXEC statement to call the program code, rather than putting the code inline.
- Use a variable reference to specify the special characters.

For example, the following Perl function contains characters representing a COMMENT block delimiter, %{, as part of its Perl language statements:

```
%FUNCTION(DTW_PERL) func() {
  ...
  for $num_words (sort bynumber keys %{ $Rtitles{$num} }) {
    &make_links($Rtitles{$num}{$num_words});
  }
  ...
%}
```

To ensure that Net.Data interprets the %{ characters as Perl source code rather than as a Net.Data COMMENT block delimiter, rewrite the function in either of the following ways:

- Use the %EXEC statement:

```
%FUNCTION(DTW_PERL) func() {
  %EXEC{ func.prl %}
%}
```

- Use a variable reference to specify the %{ characters:

```
%define percent_openbrace = "%{"

%FUNCTION(DTW_PERL) func() {
  ...
  for $num_words (sort by number keys $(percent_openbrace) $Rtitles{$num} } {
    &make_links($Rtitles{$num}{$num_words});
  }
  ...
%}
```

Message Blocks

The MESSAGE block lets you determine how to proceed after a function call, based on the success or failure of the function call, and lets you display information to the caller of the function. When processing a message, Net.Data sets the language environment variable RETURN_CODE for each function call to a FUNCTION block. RETURN_CODE is not set on a function call to a MACRO_FUNCTION block.

A MESSAGE block consists of a series of message statements, each of which specifies a return code value, message text, and an action to take. The syntax of a MESSAGE block is shown in the language constructs chapter of the *Net.Data Reference*book.

A MESSAGE block can have a global or a local scope. If it is specified at the outermost macro layer, the MESSAGE block has global scope and is active for all function calls executed in the Net.Data macro. If you define more than one global MESSAGE block, the last one defined is active. However, if the MESSAGE block is defined in a FUNCTION block, its scope is local to that FUNCTION block (except for Net.Data built-in functions, whose errors are handled by global message blocks).

Net.Data uses these rules to process the value of the RETURN_CODE or SQL_STATE variables from a function call:

1. Check the local MESSAGE block for an exact match of the value of the RETURN_CODE or SQL_STATE; exit or continue as specified.
2. If the value is not 0, check local MESSAGE block for `+default` or `-default`; depending on the sign of the value, exit or continue as specified.
3. If the value is not 0, check local MESSAGE block for `default`; exit or continue as specified.
4. Check global MESSAGE block for an exact match of the RETURN_CODE or SQL_STATE; exit or continue as specified.
5. If the value is not 0, check global MESSAGE block for `+default` or `-default`; depending on the sign of the value, exit or continue as specified.
6. If the value is not 0, check global MESSAGE block for `default`; exit or continue as specified.
7. If the value is not 0, issue Net.Data internal default message and exit.

The following example shows part of a Net.Data macro with a global MESSAGE block and a MESSAGE block for a function.

```
%{ global message block %}
%MESSAGE {
   -100     : "Return code -100 message"   : exit
    100     : "Return code 100 message"    : continue
   +default : {
This is a long message that spans more
```

```
than one line. You can use HTML tags, including
links and forms, in this message. %}   : continue
%}

%{ local message block inside a FUNCTION block %}
%FUNCTION(DTW_REXX) my_function() {
  %EXEC { my_command.mbr %}
  %MESSAGE {
     -100     : "Return code -100 message"   : exit
      100     : "Return code 100 message"    : continue
     -default : {
This is a long message that spans more
than one line. You can use HTML tags, including
links and forms, in this message. %}   : exit
  %}
```

If *my_function()* returns with a RETURN_CODE value of 50, Net.Data processes the error in this order:

1. Check for an exact match in the local MESSAGE block.
2. Check for `+default` in the local MESSAGE block.
3. Check for `default` in the local MESSAGE block.
4. Check for an exact match in the global MESSAGE block.
5. Check for `+default` in the global MESSAGE block.

When Net.Data finds a match, it sends the message text to the Web browser and checks the requested action.

When you specify `continue`, Net.Data continues to process the Net.Data macro after printing the message text. For example, if a macro calls `my_functions()` five times and error 100 is found during processing with the MESSAGE block in the example, output from a program can look like this:

```
.
.
.
11 May 1997                  $245.45
13 May 1997                  $623.23
19 May 1997                  $ 83.02
return code 100 message
22 May 1997                  $ 42.67

Total:                       $994.37
```

Calling Functions

Use a Net.Data function call statement to call both user-defined functions and built-in functions. Use the at (@) character followed by a function name or a macro function name:

`@`*`function_name`*`([ `*`argument`*`,... ])`

function_name

This is the name of the function or macro function to invoke. The function must already be defined in the Net.Data macro, unless this is a built-in function.

argument

This is the name of a variable, a quoted string, a variable reference, or a function call. Arguments on a function call are matched up with the parameters on a function or macro function parameter list. And, each parameter is assigned the value of its corresponding argument while the function or macro function is being processed. The arguments must be the same number and type as the corresponding parameters.

Quoted strings as arguments can contain variable references and functions calls.

Example 1: Function call with a text string argument

```
@myFunction("abc")
```

Example 2: Function call with a variable and a function call arguments

```
@myFunction(myvar, @DTW_rADD("2","3"))
```

Example 3: Function call with a text string argument that contains a variable reference and a function call

```
@myFunction("abc$(myvar)def@DTW_rADD("2","3")ghi")
```

Calling Net.Data Built-in Functions

Net.Data provides a large set of built-in functions to simplify Web page development. These functions are already defined by Net.Data, so you do not need to define them. You can call these functions as you would call other functions.

Figure 10 on page 82 shows how the Net.Data built-in functions and the macro interact.

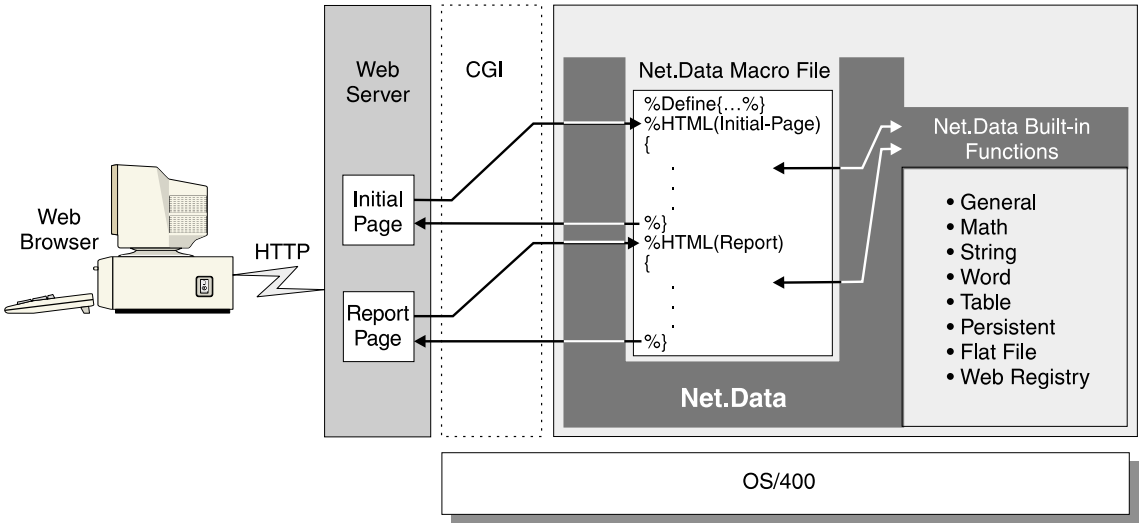

Figure 10. Net.Data Built-in Functions

Built-in functions can return their results in three ways, depending on its prefix:

- **DTW_, DTWF_, and DTWR_:** The results of the call are returned in an output parameter or no result is returned. (**DTWF_** is the prefix for flat file functions. **DTWR_** is the prefix for Web registry functions.)
- **DTW_r and DTWR_r:** The results of the function call replace the function call in the macro, in the same way the value of the RETURNS keyword replaces the function call for a user-defined function which has specified a RETURNS keyword.
- **DTW_m:** Multiple results are returned in each of the parameters passed to the function.

Some built-in functions do not have each type. To determine which type a particular built-in function has, see the Net.Data built-in functions chapter in *Net.Data Reference.*

The following sections provide a high-level overview of the Net.Data built-in functions. Use these functions to perform general purpose, math, string, word, or table manipulation functions. Some of these functions require variables to be set prior to their use or must be used in a specific context. See *Net.Data Reference* for descriptions of each function with syntax and examples.

- "General Purpose Functions" on page 83
- "Math Functions" on page 84
- "String Functions" on page 84
- "Word Functions" on page 84

General Purpose Functions

This set of functions helps you develop Web pages by altering data or accessing system services. You can use them to send mail, process HTTP cookies, generate HTML escape codes, and get other useful information from the system.

For example, to specify that Net.Data should exit a macro if a specific condition occurs, without processing the rest of the macro, you use the DTW_EXIT function:

```
%HTML(sort_page) {

<html>
 <head>
 <title>This is the page title</title>
 </head>
 <body>
 <center>
 <h3>This is the Main Heading</h3>
 <!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!>
 <! Joe Smith sees a very short page                    !>
 <!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!>
 %IF (customer == "Joe Smith")
</body>
</html>

@DTW_EXIT()

%ENDIF

...

</body>
 </html>
 %}
```

Another useful function is the DTW_URLESCSEQ function, which replaces characters that are not allowed in a URL with their escape values. For example, if the input variable `string1` equals `"Guys & Dolls"`, DTW_URLESCSEQ assigns the output variable to the value `"Guys%20%26%20Dolls"`.

Math Functions

These functions perform mathematical operations, letting you calculate or alter numeric data. Besides standard mathematical operations, you can also perform modulus division, specify a result precision, and use scientific notation.

For example, the function DTW_POWER raises the value of its first parameter to the power of its second parameter and returns the result, as shown in the following example:

```
@DTW_POWER("2", "-3", result)
```

DTW_POWER returns ".125" in the variable result

String Functions

These functions let you manipulate characters within strings. You can change a string's case, insert or delete characters, assign a string value to another variable, plus other useful functions.

For example, you can use DTW_ASSIGN to assign a value or to change the value of a variable. You can also use this function to assign a value to or to change the value of a variable. In the following example, the variable RC is assigned to zero.

```
@DTW_ASSIGN(RC, "0")
```

Other string functions include DTW_CONCAT, which concatenates strings, and DTW_INSERT, which inserts strings at a specific position, as well many other string manipulation functions.

Word Functions

These functions let you manipulate words in character strings. Most of these functions work similar to string functions, but on entire words. For example, they let you count the number of words in a string, remove words, search a string for a word.

For example, use DTW_DELW0RD to delete a specified number of words from a string:

```
@DTW_DELWORD("Now is the time", "2", "2", result)
```

DTW_DELWORD returns the string "Now time".

Other word functions include DTW_WORDLENGTH, which returns the number of characters in a word, and DTW_WORDPOS, which returns the position of a word within a string.

Table Functions

You can use these functions to generate reports or forms using the data in a Net.Data table variable. You can also use these functions to create Net.Data tables, and to manipulate and retrieve values in those tables. Table variables contain a set of values and their associated column names. They provide a convenient way to pass groups of values to a function.

For example, DTW_TB_APPENDROW appends a row to the table. In the following example, Net.Data appends ten rows to the table, `myTable`:

```
@DTW_TB_APPENDROW(myTable, "10")
```

Additionally, DTW_TB_DUMPH, returns the contents of a macro table variable, enclosed in <pre></pre> tags, with each row of the table displayed on a different line. And DTW_TB_CHECKBOX returns one or more HTML check box input tags from a macro table variable.

Flat File Functions

Use the flat file interface (FFI) functions to open, read, and manipulate data from flat file sources (text files), as well as store data in flat files.

For example, DTWF_APPEND, writes the contents of a table variable to the end of a file, and DTWF_DELETE deletes records from a file.

Additionally, the FFI functions allow file locking with DTWF_CLOSE and DTWF_OPEN. DTWF_OPEN locks a file that so that another request cannot read or update the file. DTWF_CLOSE releases the file when Net.Data is done with it, allowing other requests to access the file.

Java Applet Functions

Use the Java Applet functions to easily generate <applet> and <parm> tags to your web page based on Net.Data variables.

For example, if you have an applet named `myApplet`, and you want to pass some parameters to the applet, including a table variable, you could do the following:

```
%define REMOTE_USER = %ENVVAR
 %define myTable = %TABLE(all)
 ...
 %HTML(report) {
 ...
 @DTWA_myApplet(REMOTE_USER, myTable)
 ...
 %}
```

This would tell Net.Data to generate an <applet> tag, as well as a <parm> tag for each of the values in the table and for the value of the REMOTE_USER environment variable.

In addition, you can pass a single column of a table. For example:

```
@DTWA_myApplet(REMOTE_USER, DTW_COLUMN(mycol)myTable)
```

This example passes the `mycol` column of the Net.Data table variable `myTable`.

Web Registry Functions

Use the Web registry functions to maintain registries and the entries they contain. A Web registry is a file with a key maintained by Net.Data to allow you to add, retrieve, and delete entries easily.

For example, DTWR_ADDENTRY adds entries, while DTWR_DELENTRY deletes entries. DTWR_LISTSUB returns information about the registry entries in an OUT table parameter, and DTWR_UPDATEENTRY replaces the existing values for a specified registry entry with a new value.

Persistent Functions

The persistent macro functions support transaction processing in Net.Data by helping you define which macro blocks are persistent within a single transaction. Use these functions to define the start and end of a transaction, which HTML blocks are persistent throughout the transaction, the scope of the variables within the transaction, and whether to commit or rollback changes within the transaction.

For example, DTW_ACCEPT identifies the transaction handle for a transaction, while DTW_TERMINATE identifies the final HTML block in the transaction. DTW_RTVHANDLE generates a unique transaction handle for blocks in the transaction. You can use DTW_COMMIT and DTW_ROLLBACK to initiate commits and rollbacks during the transaction.

See “Chapter 7. Transaction Management with Persistent Macros” on page 133 for more information. Also see the built-in functions chapter in *Net.Data Reference* for a list of valid persistent functions with syntax and examples.

Generating Document Markup

Net.Data dynamically generates HTML or XML documents to be used by a client application such as a Web browser. The following sections describe the various constructs you can use to format documents with Net.Data macros. See the language constructs chapter in *Net.Data Reference* for the specific syntax information for each.

HTML and XML Blocks

The client application invokes Net.Data by specifying both the macro name and the name of one of the macro’s entry points. The entry point to the macro

can be either an HTML or XML block. These blocks contain the Net.Data language statements and text presentation statements that generate the resulting page.

Because the entry point block drives the execution of the macro, at least one entry point must exist in a macro. There can be multiple HTML or XML blocks, but only one is executed per client request. And, with each request a single document is returned to the client. To create an application consisting of many client documents, you can invoke Net.Data multiple times to process various HTML or XML blocks in various macros using standard navigation techniques, such as links and forms.

Any text presentation statements can appear in an HTML or XML block, as long as the statements are valid for the client. For example, HTML blocks can contain HTML or JavaScript. The JavaScript is not executed by Net.Data, but is sent along with the rest of the output to the client for execution and display. In an HTML or XML block, you can also include function calls, variable references, and INCLUDE statements. The following example shows a common use of an HTML block in a Net.Data macro:

```
%HTML(input){
<h1>Hardware Query Form</h1>
<hr/>
<form method="post" action="report">
<dl>
</dt>What hardware do you want to list?
<dd><input type="radio" name="hdware" value="MON" checked />Monitors</ddl>
<dd><input type="radio" name="hdware" value="PNT" />Pointing devices</ddl>
<dd><input type="radio" name="hdware" value="PRT" />Printers</ddl>
<dd><input type="radio" name="hdware" value="SCN" />Scanners</ddl>
</dl>
<hr />
<input type="submit" value="Submit" />
</form>
%}

%FUNCTION(DTW_SQL) myQuery() {
SELECT MODNO, COST, DESCRIP FROM EQPTABLE WHERE TYPE='$(hardware)'
%REPORT{
<b>Here is the list you requested:</b><br />
%ROW{
<hr />
$(N1): $(V1)    $(N2): $(V2)
</p>
$(V3)
%}
%}
%}

%HTML(report){
@myQuery()
%}
```

You can invoke the Net.Data macro from an HTML link.

```
<a href="http://www.ibm.com/cgi-bin/db2www/equiplst.dtw/input">
  List of hardware</a>
```

When the application user clicks on this link, the Web browser invokes Net.Data, and Net.Data parses the macro. When Net.Data begins processing the HTML block specified on the invocation, in this case input, it begins to process the text inside the block. Anything that Net.Data does not recognize as a Net.Data macro language construct, it sends to the browser for display.

After the user makes a selection and presses the Submit button, the client requests the action specified in the action attribute of the HTML form. This action specifies a call to the output HTML block of the macro. Net.Data then processes the output HTML block, just as it did with the input HTML block.

Net.Data then processes the `myQuery()` function call, which in turn invokes the SQL Language Environment FUNCTION block. After replacing the `$(hdware)` variable reference in the SQL statement with the value returned from the input form, Net.Data runs the query. At this point, Net.Data resumes processing the report, displaying the results of the query according to the text presentation statements specified in the REPORT block.

After Net.Data completes the REPORT block processing, it returns to the output HTML block, and finishes processing.

Report Blocks

Use the REPORT block language construct to format and display data output from a FUNCTION block. This output is typically table data, although any valid combination of text, macro variable references, and function calls can be specified. A table name can optionally be specified on the REPORT block. If you do not specify a table name, Net.Data uses the table data from the first output table in the FUNCTION parameter list.

The REPORT block has three parts, each of which is optional:

- Header information, which contains text that is displayed once before the table row data.
- A ROW block, which contains text and table variables that are displayed once for each row of the result table.
- Footer information, which contains text that is displayed once after the table row data.

Example:

```
%REPORT{
<h2>Query Results</h2>
<p>Select a name for details.
<table border=1>
```

```
  <tr>
    <td>Name</td>
    <td>Location</td></tr>
  %ROW{
  <tr>
    <td>
<a href="/cgi-bin/db2www/name.dtw/details?name=$(V1)&loc=$(V2)">$(V1)</a>
    </td>
    <td>$(V2)</td>
  </tr>
  %}
</table>
%}
```

REPORT Block Guidelines

Use the following guidelines when creating REPORT blocks:

- To avoid displaying any table output from the ROW block, leave the ROW block empty or omit it entirely.
- Use Net.Data-provided variables inside the REPORT block to access the data in the Net.Data macro results table. These variables are described in "Table Processing Variables" on page 73. For additional detail, see the Report Variables section in the *Net.Data Reference*.
- To provide header and footer information, provide the text before and after the ROW block. Net.Data processes everything it finds before a ROW block as header information. Net.Data processes everything it finds after the ROW block as footer information. As with the HTML block, Net.Data treats everything in the header, ROW, and footer blocks that is not recognized as macro language constructs as text presentation statements and sends these statements to the browser.
- You can call functions and reference variables in a REPORT block.
- To have Net.Data print a default report using pre-formatted text, do not include the REPORT block in the macro. The following example shows the default report format when the function is called in an HTML block:

```
 SHIPDATE   | RECDATE    | SHIPNO   |
------------------------------------
 25/05/1997 | 30/05/1997 | 1495194B |
------------------------------------
 25/05/1997 | 28/05/1997 | 2942821G |
------------------------------------
```

- To use the HTML tags instead of the pre-formatted text, set DTW_HTML_TABLE to YES.
- To disable the printing of the a default report, set DTW_DEFAULT_REPORT to NO or by specifying an empty REPORT block. For example:

```
%REPORT{%}
```

Example: Customizing a Report

The following example shows how you can customize report formats using special variables and HTML tags. It displays the names, phone numbers, and FAX numbers from the table CustomerTbl:

```
%DEFINE SET_TOTAL_ROWS="YES"
...
  %FUNCTION(DTW_SQL) custlist() {
    SELECT Name, Phone, Fax FROM CustomerTbl
    %REPORT{
<i>Phone Query Results:</i>
<br />
====================
<br />
    %ROW{
Name: <b>$(V1)</b>
<br />
Phone: $(V2)
<br />
Fax: $(V3)
<br />
------------------------------
<br />
    %}
    Total records retrieved: $(TOTAL_ROWS)
  %}
%}
```

The resulting report looks like this in the Web browser:

```
Phone Query Results:
====================
Name: Doen, David
Phone: 422-245-1293
Fax: 422-245-7383
------------------------------
Name: Ramirez, Paolo
Phone: 955-768-3489
Fax: 955-768-3974
------------------------------
Name: Wu, Jianli
Phone: 525-472-1234
Fax: 525-472-1234
------------------------------
Total records retrieved: 3
```

Net.Data generated the report by:

1. Printing *Phone Query Results:* once at the beginning of the report. This text, along with the separator line, is the header part of the REPORT block.
2. Replacing the variables V1, V2, and V3 with their values for Name, Phone, and Fax respectively for each row as it is retrieved.
3. Printing the string *Total records retrieved:* and the value for TOTAL_ROWS once at the end of the report. (This text is the footer part of the REPORT block.)

Multiple REPORT Blocks

You can specify multiple REPORT blocks within a single FUNCTION or MACRO FUNCTION block to generate multiple reports with one function call.

Typically, you would use multiple REPORT blocks with the DTW_SQL language environment with a function that calls a stored procedure, which returns multiple result sets (see "Stored Procedures" on page 122). However, multiple REPORT blocks can be used with any language environment to generate multiple reports.

To use multiple REPORT blocks, place a Net.Data table variable in the function parameter list. If more result sets are returned from the stored procedure than the number of REPORT blocks you have specified, and if the Net.Data built-in function DTW_DEFAULT_REPORT = ″MULTIPLE″, then default reports are generated for each table that is not associated with a report block. If no report blocks are specified, and if DTW_DEFAULT_REPORT = ″YES″, then only one default report will be generated. Note that for the SQL language environment only, a DTW_DEFAULT_REPORT value of ″YES″ is equivalent to a value of ″MULTIPLE″.

Examples: The following examples demonstrate ways in which you can use multiple report blocks.

To display multiple reports using default report formatting:

Example 1: DTW_SQL language environment

```
%DEFINE DTW_DEFAULT_REPORT = "MULTIPLE"
%FUNCTION (dtw_sql) myStoredProc (OUT table1, table2) {
     CALL myproc   %}
```

In this example, the stored procedure `myproc` returns two result sets, which are placed in `table1` and `table2`. Because no REPORT blocks are specified, default reports are displayed for both tables, `table1` first, then `table2`.

Example 2: MACRO_FUNCTION block. In this example, two tables are passed into the MACRO_FUNCTION block. When DTW_DEFAULT_REPORT=″MULTIPLE″ is specified, Net.Data generates reports for both tables.

```
%DEFINE DTW_DEFAULT_REPORT = "MULTIPLE"
%MACRO_FUNCTION multReport (INOUT tablename1, tablename2) {
%}
```

In this example, two tables are passed into the MACRO_FUNCTION `multReport`. Again, Net.Data displays default reports for the two tables in the order in which they appear in the MACRO FUNCTION block parameter list, `table1` first, then `table2`.

Example 3: DTW_REXX language environment

```
%DEFINE DTW_DEFAULT_REPORT = "YES"
%FUNCTION (dtw_rexx) multReport (INOUT table1, table2) {
     SAY 'Generating multiple default reports...<br />'
%}
```

In this example, two tables are passed into the REXX function `multReport`. Because DTW_DEFAULT_REPORT=″YES″ is specified, Net.Data displays a default report for the first table only.

To display multiple reports by specifying REPORT blocks for display processing:

Example 1: Named REPORT blocks

```
%FUNCTION(dtw_sql) myStoredProc (OUT table1, table2) {
   CALL myproc (table1, table2)

   %REPORT(table2) {
     ...
     %ROW {  ....   %}
     ...
   %}

   %REPORT(table1) {
     ...
     %row {  ....   %}
     ...
   %}
%}
```

In this example, REPORT blocks have been specified for both of the tables passed in the FUNCTION block parameter list. The tables are displayed in the order they are specified on the REPORT blocks, `table2` first, then `table1`. By specifying a table name on the REPORT blocks and the CALL statement, you can control the order in which the reports are displayed.

Example 2: Unnamed REPORT blocks

```
%FUNCTION(dtw_sql) myStoredProc  (OUT table1, table2) {
   CALL myproc

   %REPORT {
     ...
     %ROW {  ....   %}
     ...
   %}
   %REPORT {
     ...
```

```
    %ROW {  ....   %}
    ...
  %}
%}
```

In this example, REPORT blocks have been specified for two result sets returned from myproc. Because there are no table names specified on the REPORT blocks, the REPORT blocks are executed for the first two result sets in the order in which they are returned from the stored procedure.

To display multiple reports using a combination of default reports and REPORT blocks:

Example: A combination of default reports and REPORT blocks

```
%DEFINE DTW_DEFAULT_REPORT = "MULTIPLE"
%FUNCTION(dtw_system) editTables (INOUT table1, table2, table3) {
  %EXEC{ /qsys.lib/mylib.lib/mypgm.pgm %}
  %REPORT(table2) {
    ...
    %ROW {  ....   %}
    ...
  %}
%}
```

In this example, only one REPORT block is specified, and because it specifies a table name of `table2`, it uses this table to display its report. Because there are fewer REPORT blocks specified the number of result sets returned from the stored procedure, default reports are displayed for the remaining for the remaining result sets: first, a default report for `table1`; then a default report for `table3`.

Guidelines and Restrictions for Multiple REPORT Blocks: Use the following guidelines and restrictions when specifying multiple REPORT blocks in a FUNCTION or MACRO_FUNCTION block.

Guidelines:

- You can specify one REPORT block per result set.
- Specify REPORT blocks for multiple tables in the order in which you want them to be processed.
- To specify default processing when there is not a REPORT block specified for a table, define DTW_DEFAULT_REPORT = ″MULTIPLE″. When Net.Data builds the Web page, it displays default reports for tables after it displays the reports for tables having REPORT blocks. Note that setting DTW_DEFAULT_REPORT = ″YES″ will result in the generation of a default report for one table only, when a REPORT block has not been specified. The exception is in the SQL language environment, where a value of YES will result in the same processing as MULTIPLE.

- To prevent Net.Data from displaying tables that do not have REPORT blocks, set DTW_DEFAULT_REPORT = ″NO″.
- When using the DTW_SAVE_TABLE_IN variable with a function that returns more than one result set, the first result set returned from the function is assigned to the DTW_SAVE_TABLE_IN table.
- Multiple report blocks can be used with any language environment.

Restrictions:

- The values of all report variables in a function, such as START_R_N and RPT_M_R, apply to all the REPORT blocks in that function. You cannot modify the value of a report variable for individual REPORT blocks.
- The MESSAGE block must be located either before or after a list of REPORT blocks, and not between REPORT blocks.
- If the first report block specifies a table name, then all report blocks must specify table names.
- If the first report block does not specify a table name, then none of the report blocks can specify table names.

Conditional Logic and Looping in a Macro

Net.Data lets you incorporate conditional logic and looping in your Net.Data macros using the IF and WHILE blocks.

IF and WHILE blocks use a condition list that helps you test one or more conditions, and then to perform a block of statements based on the outcome of the condition test. The condition list contains logical operators, such as = and <=, and terms, which are made up of quoted strings, variables, variable references, and function calls. Quoted strings can contain variable references and functions calls, as well. You can nest the condition list.

The following sections describe conditional logic and looping:

- “Conditional Logic: IF Blocks”
- “Looping Constructs: WHILE Blocks” on page 97

Conditional Logic: IF Blocks

Use the IF block for conditional processing in a Net.Data macro. The IF block is similar to IF statements in most high-level languages because it provides the ability to test one or more conditions, and then to perform a block of statements based on the outcome of the condition test.

You can specify IF blocks almost anywhere in a macro and can nest them. The syntax of an IF block is shown in the language constructs chapter in *Net.Data Reference*.

IF Block Rules: The rules for IF block syntax are determined by the block's position in the macro. The elements allowed in the executable block of statements of an IF block depend on the location of the IF block itself.

- Any element that is valid in the block containing the IF block is valid within that IF block. For example, if you specify an IF block inside an HTML block, any element that is allowed in the HTML block is allowed in the IF block, such as INCLUDE statements and WHILE blocks.

```
%HTML block
...
  %IF block
...
    %INCLUDE
...
    %WHILE
...
  %ENDIF
%}
```

- Similarly, if you specify the IF block outside of any other block in the declaration part of the Net.Data macro, only those elements allowed outside of any other block (such as a DEFINE block or FUNCTION block) are allowed in the IF block.

```
%IF
...
  %DEFINE
...
  %FUNCTION
...
%ENDIF
```

- When an IF block is nested within an IF block that is outside of any other block in the declaration part, it can use any element that the outside block can use. When an IF block is nested within another block that is in an IF block, it takes on the syntax rules for the block it is inside.

  For example, a nested IF block must follow the rules used when it is inside an HTML block.

```
%IF
...
  %HTML {
...
     %IF
...
     %ENDIF
  %}
...
%ENDIF
```

  Exception: Do not specify a ROW block in an IF block.

IF Block String Comparison

Net.Data processes the IF block condition list in one of two ways based on the contents of the terms making up the conditions. The default action is to treat all terms as strings, and to perform string comparisons as specified in the conditions. However, if the comparison is between two strings representing integers, then the comparison is numeric. Net.Data assumes a string is numeric if it contains only digits, optionally preceded by a '+' or '-' character. The string cannot contain any non-digit characters other than the '+' or '-'. Net.Data does not support numerical comparison of non-integer numbers.

Examples of valid integer strings:

```
+1234567890
-47
000812
92000
```

Examples of invalid integer strings:

```
-  20     (contains blank characters)
234,000   (contains a comma)
57.987    (contains a decimal point)
```

Net.Data evaluates the IF condition at the time it executes the block, which can be different than the time it is originally read by Net.Data. For example, if you specify an IF block in a REPORT block, Net.Data does not evaluate the condition list associated with the IF block when it reads the FUNCTION block definition containing the REPORT block, but rather when it calls the function and executes it. This is true for both the condition list part of the IF block and the block of statements to be executed.

IF Block Example: A macro containing IF blocks inside other blocks

```
%{ This macro is called from another macro, passing the operating system
   and version variables in the form data.
%}

%IF (platform == "AS400")
  %IF (version == "V3R2")
     %INCLUDE "as400v3r2_def.hti"
  %ELIF (version == "V3R7")
     %INCLUDE "as400v3r7_def.hti"
  %ELIF (version == "V4R1")
     %INCLUDE "as400v4r1_def.hti"
  %ENDIF
%ELSE
     %INCLUDE "default_def.hti"
%ENDIF

%MACRO_FUNCTION numericCompare(IN term1, term2, OUT result) {
  %IF (term1 < term2)
    @dtw_assign(result, "-1")
  %ELIF (term1 > term2)
    @dtw_assign(result, "1")
  %ELSE
```

```
    @dtw_assign(result, "0")
  %ENDIF
%}

%HTML(report){
  %WHILE (a < "10") {
    outer while loop #$(a)<br />
    %IF (@dtw_rdivrem(a,"2") == "0")
      this is an even number loop<br />
    %ENDIF
    @DTW_ADD(a, "1", a)
  %}
%}
```

Looping Constructs: WHILE Blocks

Use the WHILE block to perform looping in a Net.Data macro. Like the IF block, the WHILE block provides the ability to test one or more conditions, and then to perform a block of statements based on the outcome of the condition test. Unlike the IF block, the block of statements can be executed any number of times based on the outcome of the condition test.

You can specify WHILE blocks inside HTML blocks, REPORT blocks, ROW blocks, MACRO_FUNCTION blocks, and IF blocks, and you can nest them. The syntax of a WHILE block is shown in the language constructs chapter of *Net.Data Reference.*

Net.Data processes the WHILE block exactly the same way it processes the IF block, but re-evaluates the condition after each execution of the block. And, like any conditional looping construct, it is possible for processing to go into an infinite loop if the condition is coded incorrectly.

Example: A macro with a WHILE block

```
%DEFINE loopCounter = "1"

%HTML(build_table) {
  %WHILE (loopCounter <= "100") {
    %{ generate table tag and column headings %}
    %IF (loopCounter == "1")
      <table border>
      <tr>
      <th>Item #
      <th>Description
    %ENDIF

    %{ generate individual rows %}
    <tr>
    <td>$(loopCounter)
    <td>@getDescription(loopCounter)

    %{ generate end table tag %}
```

```
    %IF (loopCounter == "100")
    %ENDIF

    %{ increment loop counter %}
    @DTW_ADD(loopCounter, "1", loopCounter)
  %}
%}
```

Chapter 6. Using Language Environments

Net.Data supplies language environments that you use to access data sources and to execute application programs containing business logic. For example, the SQL language environment lets you pass SQL statements to a DB2 database, and the REXX language environment lets you invoke REXX programs. You can also use the SYSTEM language environment to execute a program or issue a command.

With Net.Data, you can add user-written language environments in a pluggable fashion. Each user-written language environment must support a standard set of interfaces that are defined by Net.Data and must be implemented as a service program. For complete details on how to create a user-written language environment, see the *Net.Data Language Environment Interface Reference.*

Figure 11 shows the relationship between the Web server, Net.Data, and the Net.Data language environments.

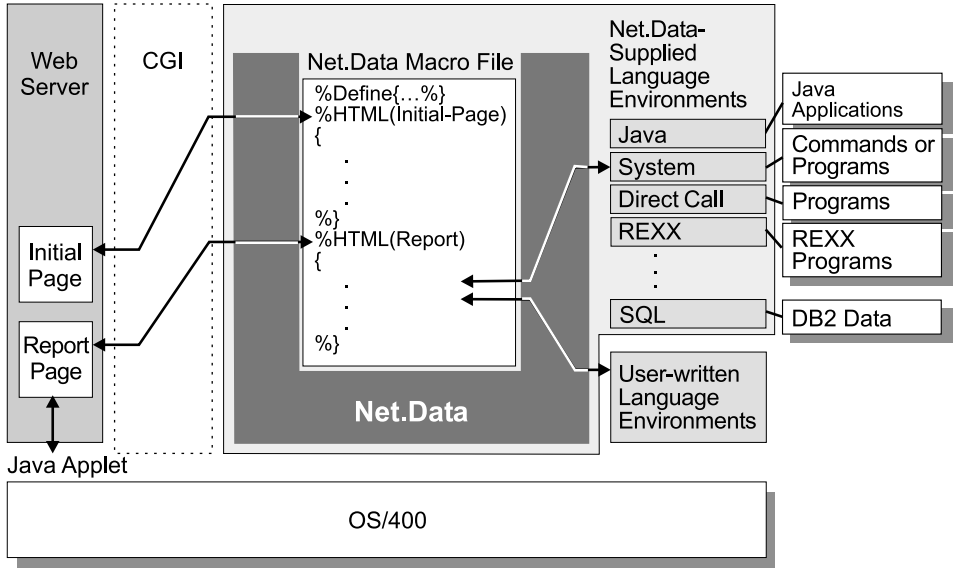

Figure 11. The Net.Data Language Environments

The following sections describe the Net.Data language environments and how to use them in your macros:

- "Overview of Net.Data-Supplied Language Environments" on page 100

© Copyright IBM Corp. 1997, 2001

- “Calling a Language Environment”
- “Direct Call Language Environment” on page 101
- “Java Application Language Environment” on page 106
- “REXX Language Environment” on page 107
- “SQL Language Environment” on page 113
- “System Language Environment” on page 130

For information about improving performance when using the language environments, see “Optimizing the Language Environments” on page 145.

Overview of Net.Data-Supplied Language Environments

Net.Data provides language environments that let you access data and programming resources for your application.

Table 3 provides a brief description of each language environment.

Table 3. Net.Data Language Environments

Language Environment	Environment Name	Description
Direct Call	DTW_DIRECTCALL	The Direct Call language environment supports calls to external programs that are written using a high-level programming language such as RPG, COBOL, and C/C++.
Java Application	DTW_JAVAPPS	Net.Data supports your existing Java applications with the Java language environment.
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. The results of the ODBC statement can be returned in a table variable.
System	DTW_SYSTEM	The System language environment supports executing commands and calling external programs.

Calling a Language Environment

To call a language environment:

- Use a FUNCTION statement to define a function that calls the language environment by supplying language statements or an %EXEC statement.

- Use a function call to the language environment.

For example:

```
%FUNCTION(DTW_SQL) custinfo() {
 select customer, custno from customer.data
 %}
 ...
 %HTML(REPORT) {
 @custinfo()
 %}
```

Guidelines for Handling Error Conditions

When an error is detected in a language environment function, the language environment sets the Net.Data RETURN_CODE variable with an error code.

You can use the following resources to handle error conditions:

- The Net.Data-supplied language environments return error codes that are documented in *Net.Data Messages and Codes Reference*.
- The database language environments, such as the SQL language environment set the RETURN_CODE variable to the SQLCODE returned by the database, and the SQL_STATE variable to the SQLSTATE returned by the database. See the messages and codes documentation for your DBMS to learn more about the SQLCODEs and SQLSTATEs used by your DBMS.

Security

Ensure that the user ID Net.Data is running under has the proper authority to access any object that may be referenced by a language environment statement. For example, SQL language environment executes SQL statements, so that the user ID under which Net.Data executes must have the authority to access the database resources, in order to execute successfully.

Direct Call Language Environment

The Direct Call language environment allows you to call programs that are written in a high-level language such as C, RPG, COBOL, and CL. Parameters can be passed *to* the program, and parameter values can be received *from* the program, enabling easy integration of existing programs with Net.Data and allowing users to use existing programming skills to code complicated business logic.

Calling Programs

To call a program, define a function that uses the Direct Call (DTW_DIRECTCALL) language environment and that includes a path to the program that is to be called in an EXEC statement. For example:

```
%function(DTW_DIRECTCALL) dc1() {
   %EXEC { /QSYS.LIB/NETDATA.LIB/MYPGM.PGM %}
%}
```

You can shorten the path to the program if you use the EXEC_PATH configuration variable to define paths to directories that contain programs. See “EXEC_PATH” on page 21 to learn how to define the EXEC_PATH configuration variable.

Supported Language Environment Variables

The Direct Call language environment supports the DTW_PAD_PGM_PARMS variable, which indicates whether parameters that are to be passed to a program are to padded with blanks up to the precision specified. See *Net.Data Reference* for description, syntax, and examples for this variable. See “Passing Parameters to Programs” for more information on passing parameters to programs.

Passing Parameters to Programs

Pass parameters to a program by specifying on the function definition the data type of the parameter and whether the parameter to be passed is an input-only (IN), output-only (OUT), or input/out (INOUT) parameter. For example:

```
%function(DTW_DIRECTCALL) dc2(IN CHAR(3) p1,
                              INOUT INTEGER p2,
                              OUT DECIMAL(7,2) p3) {
   %EXEC { /QSYS.LIB/NETDATA.LIB/MYPGM.PGM %}
%}
```

In the above example, the Direct Call language environment passes three parameters, a character variable, an integer, and a packed decimal variable to the program MYPGM. You can pass up to 50 parameters to the called program. Only parameters specified with data types are passed to the program. The Direct Call language environment converts the string corresponding to the parameter to the internal representation of the data type. The language environment then passes pointers to the internal representation of the variables to the called program, in the order specified on the function definition.

Because pointers to the variables are passed to the program, the program can change the value of the variable. However, only OUT or INOUT variables that are changed by the program are reflected back in the macro that called the language environment.

Supported Data Types

Table 4 on page 103 lists the data types that are supported by the Direct Call language environment. Not all of the data types are supported by each high-level language.

Table 4. Direct Call Data Types

Data Type	Usage Notes
CHAR(n) CHARACTER(n) CHARACTER	A character string. If *n* is specified, it must be greater than zero. If the string is not specified, it is assumed to be one character. Because all strings passed from the Direct Call language environment are null-terminated, the language environment allocates n+1 bytes (1 byte for the NULL terminator). Strings that exceed *n* are truncated.
VARCHAR(*n*)	A variable-length character string, where *n* is greater than zero, and less than or equal to 32740. The string is null-terminated and the language environment allocates n+2+1 bytes (2 bytes to store the string length, 1 byte for the null-terminator). Strings that exceed *n* are truncated. The first two bytes of the string contain the string length (binary value). If the parameter is defined as OUT (output only), the string length is set to zero before the variable is passed to the called program.
INTEGER INT	A signed binary integer, 4 bytes long.
SMALLINT	A signed binary integer, 2 bytes long
FLOAT(*p*,*s*)	A single-precision or double-precision, floating point number. For single precision, *p* must be greater than 0 and less than 25. For double precision, *p* must be greater than 24 and less than 54. The precision (*p*) and scale (*s*) are only used when converting data to a displayable format; for example, to a string.
REAL(*p*,*s*)	A single-precision floating point number. *p* must be greater than 0 and less than 25. The precision (*p*) and scale (*s*) are only used when converting data to a displayable format; for example, to a string.
DOUBLE (*p*,*s*) DOUBLEPRECISION(*p*,*s*)	A double-precision floating point number. *p* must be greater than 0 and less than 53. The precision (*p*) and scale (*s*) are only used when converting data to a displayable format; for example, to a string.
NUMERIC(*p*,*s*)	A zoned decimal number, with precision *p* and scale *s*. The value of *p* must be greater than 0 and less than 32.
DEC(*p*,*s*) DECIMAL(*p*,*s*)	A packed decimal number, with precision *p* and scale *s*. The value of *p* must be greater than 0 and less than 32.
DTWTABLE	A special data type used to pass a Net.Data table to the called program. The Direct Call language environment passes a pointer to the table, which can be manipulated using the Net.Data language environment interface table functions.

Parameters that are defined to be numeric can include the currency symbol and three-digit separators. The Direct Call language environment removes the currency symbol and three-digit separators when converting a numeric variable from string form to its internal form, before passing the variable to the program. Net.Data retrieves the currency symbol, decimal format, and three-digit separator characters from the process attributes of the process in which Net.Data is running.

Null-Terminated String Parameters

If DTW_PAD_PGM_PARMS is set to NO in the configuration file or within the macro, the Direct Call language environment passes string values to your program using a null terminator character (value x'00'). This requires you to write code to handle the string (unless you are using C or C++, which expect null terminated strings).

For example, if you define the parameter field as CHAR(10), but pass a string value that is 5 bytes long, Net.Data puts the null terminator after the fifth byte. Passing the value ″12345″ as a string in a CHAR(10) field yields:

```
x'F1F2F3F4F500........'
```

The bytes following the null terminator are undefined (you cannot assume that the bytes are null or blank).

Because the string is null terminated and contains uninitialized bytes after the null terminator, you cannot use the string in an RPG or COBOL program. For example, if you use the string in a comparison operation, the operation does not yield valid results. The program does not expect the string to contain the null terminator and expects the string to be padded with blanks at the end.

You can use string handling functions within your program to extract the string value or use the VARCHAR data type. This method gives the length of the string in the first two bytes.

If DTW_PAD_PGM_PARMS is set to YES in the configuration file or within the macro, the Direct Call language environment passes string values to your program with the values padded to the right with blanks up to the precision length. Using the same example as above, but with DTW_PAD_PGM_PARMS set to YES, passing the value ″12345″ as a string in a CHAR(10) field yields:

```
x'F1F2F3F4F5404040404000'
```

Because the string has a length of 5, which is less than the specified precision, blanks are inserted after the value up to the precision length. Programs written in languages such as RPG can now use the parameter without the need to handle NULL-terminated strings.

Common Errors when Passing Parameters

The following list describes errors that can occur when calling programs and passing parameters to the program using the Direct Call language environment. Tips for avoiding these errors are provided.

Parameter Mismatch Errors

Ensure that the number and order of parameters match the number and order in which they appear in the parameter list of the called program.

Data Type Errors

Ensure that the data type specified for a parameter matches the data type expected by the called program. There might be data types supported by the Direct Call language environment that are not supported by the high level programming language used to create the called program.

Length Errors

Ensure that the lengths defined for parameters are correct and match the lengths specified in the called program. Specifying a length that is shorter than the declared length of the called program might corrupt storage and cause Net.Data not to function properly.

Returning Values from Programs

Some high level programming languages, such as C, can return an integer on the program call. The integer can be retrieved by specifying the RETURNS keyword in the function definition, For example:

```
%function(DTW_DIRECTCALL) dc3(IN CHAR(3) p1) RETURNS(retval) {
   %EXEC { /QSYS.LIB/NETDATA.LIB/MYPGM.PGM %}
%}
```

When the function call completes successfully, the parameter *retval* contains the value returned by the program.

Direct Call Language Environment Example

In this example, the macro calls a program and passes several parameters. The source for the program follows the macro, and is written in RPG and CL. The program that is called accepts two integer parameters. It copies the first parameter (the input parameter) to the second parameter (the output parameter).

Macro:

```
%define ilepgm = "/QSYS.LIB/NETDATADEV.LIB/TDCCLI01.PGM"
 %define out1  = "0"

 %FUNCTION(DTW_DIRECTCALL) dcFunction(IN INT inp1,
                                      OUT INT outp2)
 {  %EXEC { $(ilepgm) %} %}
```

```
%HTML(REPORT) {
@dcFunction("123", out1)
The value of out1 is: "$(out1)"
%}
```

ILE RPG program:

```
DINP1             S             10I00
DOUTP2            S             10I00
C*
C     *ENTRY        PLIST
C                   PARM                    INP1
C                   PARM                    OUTP2
C*
C                   Z-ADD     INP1          OUTP2
C*
C                   SETON                                        LR
```

CL program:

```
PGM PARM(&INP1; &OUTP2;)

DCL VAR(&INP1;)  TYPE(*CHAR) LEN(4)
DCL VAR(&OUTP2;) TYPE(*CHAR) LEN(4)

CHGVAR VAR(&OUTP2;) VALUE(&INP1;)
ENDPGM
```

Java Application Language Environment

The Java Application language environment allows you to call Java programs, enabling easy integration of Java applications with Net.Data. The Java Application language environment was first introduced in OS/400 V4R4.

To use the Java Application language environment, complete the configuration steps documented in “Setting up the Java Application Language Environment” on page 27.

Calling Java Programs

To call a Java program, define a function that uses the Java Application (DTW_JAVAPPS) language environment. Specifying a function name that represents the class name of the Java program.

Example: Calls a Java program `helloWorld.java`:

```
%function(DTW_JAVAPPS) helloWorld() { %}
```

The Java Application language environment expects Java programs to contain a method identifier for ’main,’ the first method that is run in a Java program. When the language environment invokes an application, the application has access to stdin and stdout. There is no form data in stdin because Net.Data has already read the data.

Important: Before calling Java applications, set the DTW_JAVA_CLASSPATH path configuration variable so that the Java class can be found. See "DTW_JAVA_CLASSPATH" on page 21 for the syntax of this variable.

Passing Parameters to Java Programs

Pass parameters to a Java program by specifying the parameters to be passed on the function definition. Specify only string parameters that are input-only (IN), or input or output (INOUT).

Example: The IN parameter p1 is to be passed on the function call

```
%function(DTW_JAVAPPS) jv1(IN p1) { %}
```

The Java Application language environment does not support Java programs that update variables because it cannot pass the updated values back to the macro.

Java Application Language Environment Example

In this example, the Net.Data macro calls a Java program, echoString. The macro passes two string parameters to the Java language environment. The first string tells the Java program whether to use italic or bold highlighting for the second parameter, a text string, before printing the second parameter to standard output (stdout). Because the program passes ″I″, for italics, the Web server displays the text string *Hello World*, in italics, at the browser. The source for the Java program follows the macro.

Macro:

```
%FUNCTION(DTW_JAVAPPS) echoString(textAttribute, text){ %}

 %HTML(runjava){
 @echoString("I","Hello World")
 %}
```

Java program:

```
class echoString {
   public static void main (String args[]) {
      if (args[0].equals("I"))
        System.out.println("<i>" + args[1] + "</i>");
      else
        System.out.println("<b>" + args[1] + "</b>");
   }
 }
```

REXX Language Environment

The REXX language environment allows you to run REXX programs written to run in the DTW_REXX environment. The Net.Data REXX Language Environment provides controls that allow REXX programs to easily return large amounts of data.

Net.Data also provides support for the REXX SAY statement that directs the output to browser regardless of what Web server environment you use for Net.Data. If you run native REXX using the Web server FastCGI, GWAPI, or Servlet configuration, the output from REXX SAY statements are routed to the Web servers log file instead of the browser. This is not true for REXX programs that are written to run in the DTW_REXX environment.

Support for Variables: To allow REXX programs to easily return large amounts of data, Net.Data automatically adds code to the beginning of the REXX program and appends code to the end of the REXX program. This code is designed to manipulate variables that were provided on the DTW_REXX function statement.

Support for REXX SAY Statements (FastCGI, GWAPI, and SERVLET environments): REXX SAY statements are automatically converted to REXX assignment statements by Net.Data prior to executing the REXX program. Net.Data automatically appends code to the REXX program that is designed to direct the output from the original REXX SAY statements to the browser. Use of REXX subroutines and functions: Since Net.Data adds code to the front of the REXX program and appends code to the end of the REXX program, the main REXX routine must end with the last statement of the REXX program. If you use REXX subroutines or functions you must insure that the last statement of the REXX program is associated with the main REXX routine.The following is an example of using a subroutine and function in a REXX program that is written to run in the DTW_REXX environment:

```
%function(DTW_REXX) genData(out s1,s2) {
         call subrtn1
         s2=funrtn1()
        signal rexxEnd  /* Go to end of Program */
      subrtn1: PROCEDURE EXPOSE s1
         string1 = "data for s1"
         return 0
      funrtn1: PROCEDURE
         retvar = "data for s2"
         return retvar
      rexxEnd:          /* End of Main Program */
         return 0
    %}
    %HTML (Report) {


    @genData(a,c)

  Value for s1: $(a)

  Value for s2: $(c)
   %}
```

Use of REXX EXIT and RETURN statements: Net.Data automatically appends code to REXX programs that provide values for output variables and directs output from SAY statements to the browser. If the REXX program issues a RETURN from the main routine or issues an EXIT statement anywhere but the last statement of the REXX program, the code that was appended by Net.Data to the REXX program will not be executed. This results in the lost of output variables and output from SAY statements. If you must exit a REXX program before reaching the last statement, you should branch to the last statement in the REXX program that normally exits. If you use the RETURN or EXIT statement to end the main REXX program, it must be the last statement in the REXX program. This includes REXX comment statements. For example:

```
%function(DTW_REXX) genData(out s1,s2) {
......
If S2 < 0 Then signal rexxEnd
......
......
rexxEnd:
/* This comment must be before the following
RETURN statement */
return 0
%}
%HTML (Report) {
@genData(a,c)
......
%}
```

Invoking external REXX programs from a DTW_REXX function: You can invoke a REXX program from a DTW_REXX function using the Net.Data %EXEC statement or from a REXX program using methods provided by REXX.

When invoking an external REXX program using the Net.Data %EXEC statement, Net.Data automatically adds code to the beginning of the REXX program and appends code to the end of the REXX program to handle Output variables and direct output from REXX SAY statements to the browser.

When you use methods provided by REXX to invoke a REXX program, Net.Data does not receive control and doesn't add code to the REXX program. The REXX program being invoked must pass output back to the calling REXX program using standard REXX conventions. When running in GWAPI or SERVLET environments, Output from REXX SAY statements are sent to the Web servers log file.

Executing REXX Programs

With the REXX language environment you can execute both in-line REXX programs or external REXX programs. An in-line REXX program is a REXX

program that has the source of the REXX program in the macro. An external REXX program has the source of the REXX program in an external file.

To execute an in-line REXX program:

Define a function that uses the REXX (DTW_REXX) language environment and contains the REXX code in the function.

Example: A function that contains a in-line REXX program

```
%function(DTW_REXX) helloWorld() {
        SAY 'Hello World'
%}
```

To run an external REXX program:

Define a function that uses the REXX (DTW_REXX) language environment and includes a path to the REXX program that is to be run in an EXEC statement.

Example: A function that contains an EXEC statement pointing to a the external program

```
%function(DTW_REXX) externalHelloWorld() {
%EXEC{ /QSYS.LIB/REXX.LIB/REXXSRC.FILE/HELLOWORLD.MBR%}
%}
```

You can shorten the path to the program if you use the EXEC_PATH configuration variable to define paths to directories that contain programs. See "EXEC_PATH" on page 21 to learn how to define the EXEC_PATH configuration variable.

Restriction: If you are running OS/400 V3R2 or V3R7 and a REXX program uses the SAY REXX instruction to write to stdout, then insert 12 blanks at the start of the string. For example:

```
SAY '            STARTOFDATA'
```

The 12 blanks are ignored, but if they are not inserted, unpredictable results might occur.

Passing Parameters to REXX programs

There are two ways to pass information to a REXX program that is invoked by the REXX (DTW_REXX) language environment, directly and indirectly.

Directly

Pass parameters directly to an external REXX program using the %EXEC statement. For example:

```
%FUNCTION(DTW_REXX) rexx1() {
  %EXEC{/QSYS.LIB/NETDATA.LIB/QREXXSRC.FILE/CALL1.MBR $(INPARM1)  %}
%}
```

The Net.Data variable INPARM1 is referenced and passed to the external REXX program. The REXX program can reference the variable by using REXX PARSE ARG instruction. The parameters that are passed to the REXX program using this method are considered input parameters, and any modification to the values are not reflected back to Net.Data. (the parameters passed to the program can be used and manipulated by the program, but changes to the parameters are not reflected back to Net.Data).

Indirectly

Pass parameters indirectly, by way of the REXX program *variable pool*. When a REXX program is started, a space which contains information about all variables is created and maintained by the REXX interpreter. This space is called the variable pool.

When a REXX language environment (DTW_REXX) function is called, any function parameters that are input (IN) or input/output (INOUT) are stored in the by the REXX language environment prior to executing the REXX program. When the REXX program is invoked, it can access these variables directly. Upon the successful completion of the REXX program, the DTW_REXX language environment determines whether there are any output (OUT) or INOUT function parameters. If so, the language environment retrieves the value corresponding to the function parameter from the variable pool and updates the function parameter value with the new value. When Net.Data receives control, it updates all OUT or INOUT parameters with the new values obtained from the REXX language environment. For example:

```
%DEFINE a = "3"
%DEFINE b = "0"
%FUNCTION(DTW_REXX) double_func(IN inp1, OUT outp1){
   outp1 = 2*inp1
%}

%HTML (Report) {
Value of b is $(b), @double_func(a, b) Value of b is $(b)
%}
```

In the above example, the call *@double_func* passes two parameters, *a* and *b*. The REXX function *double_func* doubles the first parameter and stores the result in the second parameter. When Net.Data invokes the macro, *b* has a value of 6.

You can pass Net.Data tables to a REXX program. A REXX program accesses the values of a Net.Data macro table parameter as REXX stem variables. To a REXX program, the column headings and field values are contained in variables identified with the table name and column number. For example, in the table myTable, the column headings are myTable_V.*j*, and the field values are myTable_V.*i*.*j*, where *i* is the row number and *j* is the column number. The number of rows in the table is myTable_*ROWS* and the number of columns in the table is myTable_*COLS*.

REXX Language Environment Example

The following example shows a macro that calls a REXX function to generate a Net.Data table that has two columns and three rows. Following the call to the REXX function, a built-in function, DTW_TB_TABLE(), is called to generate an HTML table that is sent back to the browser.

```
%DEFINE myTable = %TABLE
%DEFINE DTW_DEFAULT_REPORT = "NO"

%FUNCTION(DTW_REXX) genTable(out out_table) {
  out_table_ROWS = 3
  out_table_COLS = 2

  /* Set Column Headings */
  do j=1 to out_table_COLS
    out_table_N.j = 'COL'j
  end

  /* Set the fields in the row */
  do i = 1 to out_table_ROWS
    do j = 1 to out_table_COLS
      out_table_V.i.j = '[' i j ']'
    end
  end
%}

%HTML (Report)  {
 @genTable(myTable)
 @DTW_TB_TABLE(myTable)
%}
```

Results:

```
<table>
 <tr>
<th>COL1</th>
<th>COL2</th>
</tr>
<tr>
<td>[1 1]</td>
<td>[1 2],</td>
</tr>
<tr>
```

```
<td>[2 1]</td>
<td>[2 2],</td>
</tr>
<tr>
<td>[3 1]</td>
<td>[3 2],</td>
</tr>
</table>
```

SQL Language Environment

The SQL language environment allows you to run SQL statements by sending the SQL statements to a database management system (DBMS).

To use the SQL language environment, ensure you follow the configuration steps documented in “Setting Up Net.Data Language Environments” on page 27

Executing SQL Statements

You can execute any SQL statement that is supported by dynamic SQL.

To execute SQL statements, define a function that uses the SQL (DTW_SQL) language environment and contains the SQL statements in the language environment executable section of the function.

Example: An SQL function that runs an SQL SELECT statement:

```
%function(DTW_SQL) getOrders() {
   SELECT cust, custid, custorder FROM mylibrary.customers
%}
```

Commitment Control

The SQL language environment by default runs under commitment control and follows all rules for governing commitment control.

- Journal all files or tables that are accessed through the DTW_SQL, except when the SQL statement is SELECT.
- Optionally change the commitment level by specifying DTW_SQL_ISOLATION in the Net.Data initialization file. See “DTW_SQL_ISOLATION: DB2 Isolation Variable” on page 16 for details about the isolation levels that the SQL language environment supports.

For more information on transaction management, see “Managing Transactions in a Net.Data Application” on page 120.

OUT and INOUT Tables

If you specify OUT or INOUT Net.Data tables on the function definition, and the SQL statement returns result sets, the SQL language environment stores each result set in the specified tables. You can then use the table later in the macro. If an OUT table is not specified, the SQL language environment uses a default table.

Nested SQL Statements

You can call SQL functions within another SQL function. If tables are passed, then ensure you use unique table names in each of the functions; otherwise, unpredictable results might occur.

Example: Calls an SQL function from the ROW block of another SQL function

```
%define mytable1 = %TABLE
%define mytable2 = %TABLE

%FUNCTION(DTW_SQL) sql2 (IN p1, OUT t2) {
   select * from NETDATA.STAFFINF where projno='$(p1)'
 %REPORT {
  %ROW { $(N1) is $(V1) %}
 %}
%}

%FUNCTION(DTW_SQL) sql1 (OUT t1) {
   select * from NETDATA.STAFFINF
 %REPORT {
   %ROW { @sql2(V1, mytable2) %}
 %}
%}

%HTML(netcall1) { @sql1(mytable1) %}
```

Supported Language Environment Variables

The SQL language environment supports variables designed to support DB2. For example, the DATABASE variable specifies the data source that the SQL language environment connects to when executing an SQL statement. The following list specifies which variables are supported for the SQL language environment. See *Net.Data Reference* for description, syntax, and examples for these variables.

- DATABASE
- DB_CASE
- DTW_EDIT_CODES
- DTW_PAD_PGM_PARMS
- DTW_SET_TOTAL_ROWS
- LOGIN
- NULL_RPT_FIELD
- PASSWORD

- SHOWSQL
- SQL_STATE
- TRANSACTION_SCOPE

Supported Data Types

The SQL language environment supports the data types listed in Table 5

Table 5. Data Types

BLOB(1)	DOUBLE	SMALLINT
CHAR	DOUBLEPRECISION	TIME
CLOB(1)	FLOAT	TIMESTAMP
DATE	GRAPHIC	VARCHAR
DBCLOB(1)	INTEGER	VARGRAPHIC
DECIMAL	REAL	

(1) These data types cannot be passed as parameters to a stored procedure call. To learn which data types are support for stored procedures, see “Stored Procedure Syntax” on page 122

See “Data Type Considerations” on page 116 to learn about special considerations for the LOBs and DATALINK data types.

SQL Language Environment Restrictions

Consider the following restrictions when planning your environment:

- Do not use the SQL language environment if at least one of the following conditions exists:
  - A user-defined language environment is created that uses the database access class library or the SQL call level interface (CLI) and the user-defined language environment is referenced in a macro
  - An application that uses the SQL CLI will be running in the same process as Net.Data
- SQL statements in the inline statement block can be up to 32KB.
- You can use up to 50 local or remote database connections. When using multiple connection, consider the following restrictions:
  - Net.Data does not allow concurrent connections to the same remote database.
  - You cannot change login IDs after you have accessed a remote database if TRANSACTION_SCOPE=MULTIPLE, which is the default. See “Managing Transactions in a Net.Data Application” on page 120.

  See “Managing Multiple Database Connections” on page 121 for more information about these restrictions.

Data Type Considerations

The following data types supported by the SQL language environment need special consideration.

- "Using Large Objects"
- "Encoding DataLink URLs in Result Sets" on page 119

Using Large Objects

You can store large object files (LOBs) in DB2 databases and incorporate them into your dynamic Web pages by using the Net.Data SQL language environment.

When the language environment executes an SQL SELECT statement or a stored procedure that returns a LOB, it does not assign the object to a V(*n*) table processing variable or a Net.Data table field. Instead, it stores the LOB in a file that Net.Data creates and returns only the name of the file in the V(*n*) table processing variable or a Net.Data table field. In your Net.Data macro you can use the name to reference the LOB file; for example, you can create an HTML anchor element with a hypertext reference or an image element containing a URL for the file. Net.Data places the file containing the LOB in the directory specified by the DTW_LOB_DIR configuration variable, located in the Net.Data initialization file (db2www.ini). Write access to the LOB file is limited to the user ID associated with the Net.Data request that retrieved the LOB.

The file name for the LOB is dynamically constructed, and has the following form:

name[*.extension*]

Where:

name Is a dynamically generated unique string identifying the large object

extension
Is a string that identifies the type of the object. For CLOBs and DBCLOBs, the extension is .txt. For BLOBs, the SQL language environment determines the extension by looking for a signature in the first few bytes of the LOB file. Table 6 shows the LOB extensions used by the SQL language environment:

Table 6. LOB extensions used in the SQL language environment

Extension	Object Type
.bmp	bitmap image
.gif	graphical image format
.jpg	joint photographic experts group (JPEG) image
.tif	tagged image file format
.ps	postscript

Table 6. LOB extensions used in the SQL language environment (continued)

Extension	Object Type
.mid	musical instruments digital interface (midi) audio
.aif	AIFF audio
.avi	audio visual interleave audio
.au	basic audio
.ra	real audio
.wav	windows audio visual
.pdf	portable document format
.rmi	midi sequence

If the object type for the BLOB is not recognized, no extension is added to the file name.

When Net.Data returns the name of the file containing a LOB, it prefixes the file name with the string /tmplobs/ using the following syntax:

```
/tmplobs/name.[extension]
```

This prefix permits you to locate your LOB directory in a directory other than the Web server's document root directory.

To ensure that references to LOB files are correctly resolved, add the following Pass directive to your Web server's configuration file:

```
Pass    /tmplobs/*     <full_path>
```

<full_path> is the value specified for the DTW_LOB_DIR configuration variable in the Net.Data initialization file.

Planning tip: Each query that returns LOBs results in files being created in the directory specified by the DTW_LOB_DIR path configuration variable. Consider system limitations when using LOBs because they can quickly consume resources. You might want to clean up the directory periodically. It is recommended that you use DataLinks, which eliminate the need to store files in directories by the SQL language environment, resulting in better performance and the use of much less system resources.

Example: The following application uses an MPEG audio (.mpa) file. Because the SQL language environment does not recognize this file type, an EXEC variable is used to append the .mpa extension to the file name. A user of this application must click on the file name to invoke the MPEG audio file viewer.

```
%DEFINE{
lobpath = "@DTW_RGETINIDATA("DTW_LOB_DIR")"
filename = "@DTW_RREPLACE($(V3), "/tmplobs/", "", "1", "F")"
myFile=%EXEC "REN '$(lobpath)/$(filename)'  '$(filename).mpa'"
%}
```

```
%FUNCTION(DTW_SQL) queryData() {
  SELECT Name, IDPhoto, Voice FROM RepProfile
  %REPORT{
    <p>Here is the information you selected:</p>
    %ROW{

      $(myFile)
      $(V1) <img src="$(V2)">
            <a href="$(V3).mpa">Voice sample</a><p>
    %}
  %}
%}

%HTML (Report){
@queryData()
%}
```

If the RepProfile table contains information about Kinson Yamamoto and Merilee Lau, then the execution of the REPORT block will add the following HTML to the Web page being generated:

```
<p>Here is the information you selected:</p>
Kinson Yamamoto <img src="/tmplobs/p2345n1.gif">
<a href="/tmplobs/p2345n2.mpa">Voice sample</a><p>
Merilee Lau <img src="/tmplobs/p2345n3.gif">
<a href="/tmplobs/p2345n4.mpa">Voice sample</a><p>
```

The REPORT block in the previous example uses the implicit table variables V1, V2, and V3.

- The value of V1 is a person's name, which is character data.
- The value of V2 is the name of a GIF file containing the photo of the person. The image is displayed inline within the generated Web page.
- The value of V3 is a sample of the person's voice in a .mpa file. When the SQL language environment encounters an unrecognized format, such as a .mpa file, it writes the file into the directory specified in the DTW_LOB_DIR configuration variable without a file extension. This example shows how to handle this file type by adding the extension using an EXEC variable. When the variable $(V3) is resolved, it has the path /tmplobs/ added before the file name. For example, /tmplobs/sound2a. In the example, the EXEC variable renames the file using the REN command, adding the extension .mpa to the file. Before the file name can be renamed, the /tmplobs/ is removed from the file name and the full path to the file to be renamed is retrieved by using the DTW_RGETINIDATA function to retrieve the path specified in DTW_LOB_DIR. The voice sample is played when the application user clicks on Voice sample.

Access rights for LOBs:

Ensure that the user ID or user IDs under which Net.Data executes have write access to the directory specified by DTW_LOB_DIR.

Encoding DataLink URLs in Result Sets

The DataLink data type is one of the basic building blocks for extending the types of data that can be stored in database files. With DataLink, the actual data stored in the column is only a pointer to the file. This file can be any type of file; an image file, a voice recording, or a text file. DataLinks store a URL to resolve the location of the file.

The DATALINK data type requires the use of DataLink File Manager. For more information about the DataLink File Manager, see the DataLinks documentation for your operating system. Before you use the DATALINK data type, you must ensure that the Web server has access to the file system managed by the DB2 File Manager Server.

When a SQL query returns a result set with DataLinks, and the DataLink column is created with FILE LINK CONTROL with READ PERMISSION DB DataLink options, the file paths in the DataLink column contains an access token. DB2 uses the access token to authenticate access to the file. Without this access token, all attempts to access the file fail with an authority violation. However, the access token might include characters that are not usable in a URL to be returned to a browser, such as the semi-colon (;) character. For example:

```
/datalink/pics/UN1B;0YPVKGG346KEBE;baibien.jpg
```

The URL is not a valid because it contains semi-colon (;) characters. To make the URL valid, the semi-colons must be encoded using the Net.Data built-in function DTW_URLESCSEQ. However, some string manipulation must be done before applying this function because this function encodes slashes (/), as well.

You can write a Net.Data MACRO_FUNCTION to automate the string manipulation and use the DTW_URLESCSEQ function. Use this technique in every macro that retrieves data from a DATALINK data type column.

Example 1: A MACRO_FUNCTION that automates the encoding of URLs returned from DB2 UDB

```
%{ TO DO: Apply DTW_URLESCSEQ to a DATALINK URL to make it a valid URL.
    IN: DATALINK URL from DB2 File Manager column.
    RETURN: The URL with token portion is URL encoded
%}
%MACRO_FUNCTION encodeDataLink(in DLURL) {
  @DTW_rCONCAT( @DTW_rDELSTR( DLURL,
```

```
  @DTW_rADD(@DTW_rLASTPOS("/", DLURL), "1" ) ),
  @DTW_rURLESCSEQ( @DTW_rSUBSTR(DLURL,
  @DTW_rADD( @DTW_rLASTPOS("/", DLURL), "1" ) ) ) )
%}
```

After using this MACRO_FUNCTION, the URL is encoded properly and the file specified in the DATALINK column can be referenced on any Web browser.

Example 2: A Net.Data macro specifying the SQL query that returns the DATALINK URL

```
%FUNCTION(DTW_SQL)myQuery(){
  select name, DLURLCOMPLETE(picture) from myTable where name like '%river%'
  %REPORT{
    %ROW{
      <p> $(V1) <br />
      Before Encoding: $(V2) <br />
      After Encoding: @encodeDataLInk($(V2)) <br />
      Make HREF: <a href="@encodeDataLink($(V2))"> click here </a> <br /> <p>
    %}
  %}
%}
```

Note that a DataLink File Manager functions is used. The function `dlurlcomplete` returns a full URL.

Managing Transactions in a Net.Data Application

When you modify the content of a database using insert, delete, or update statements, the modifications do not become persistent until the database receives a commit statement from Net.Data. If an error occurs, Net.Data sends a rollback statement to the database, reversing all modifications since the last commit.

The way in which Net.Data sends the commit and possible rollback statements depend on the setting of TRANSACTION_SCOPE and whether commit statements are explicitly specified in the macro. The values for TRANSACTION_SCOPE are MULTIPLE and SINGLE. The default is MULTIPLE. To set TRANSACTION_SCOPE to SINGLE, use a %DEFINE statement or a call to @DTW_ASSIGN(), and pass the variable on the ENVIRONMENT statement for the proper LE. For more information, see Customizing the Net.Data Initialization File in Chapter 2 of this book.

SINGLE

Specifies that Net.Data issues a commit statement after each successful SQL statement. If the SQL statement returns an error, a rollback statement is issued. SINGLE transaction scope secures a database

modification immediately; however, with this scope, it is not possible to undo a modification using a rollback statement later.

MULTIPLE

Specifies that Net.Data will execute all SQL statements before a commit statement is issued. Net.Data sends the commit at the end of the request, and if each SQL statement is issued successfully, the commit makes all modifications in the database persistent. If any of the statements returns an error, Net.Data issues a rollback statement at the point of the error, which sets the database back to its prior state.

By leaving TRANSACTION_SCOPE set to MULTIPLE and issuing commit statements at the end of those groups of statements that you feel qualify as a transaction, you the application developer maintain full control over the commit and rollback behavior in your application.

To issue an SQL commit statement, you can define a function that you can call in at any point in your HTML block:

```
%FUNCTION(DTW_SQL) user_commit() {
  commit
%}

...

%HTML {
  ...
  @user_commit()
  ...
%}
```

Restrictions:

The setting of TRANSACTION_SCOPE cannot be changed after a connection to the database is made. Therefore, all SQL transactions in a macro are subject to the same processing.

If you are using Net.Data as part of Net.Commerce, note that Net.Commerce has its own transaction handling and disables the transaction handling of Net.Data.

Managing Multiple Database Connections

You can connect to up to 50 local or remote databases at a time. The SQL language environment keeps the connections active for the life of the Web server process job that Net.Data is running under. Keeping the connections active provides fast database access after the initial connection to the database. You can prevent errors by taking the following issues into consideration:

- Net.Data does not allow concurrent connections to the same remote database. If a connection exists to a remote database using one user ID (the

LOGIN SQL language environment parameter) and another request is made to connect to the same remote database using a second user ID, the SQL language environment must first disconnect the existing connection, do a commit (if commitment control is being used) and then reestablish the connection using the 'new' user ID and password. The commit is required because if the connection is broken, there is no way that a rollback can be accomplished in case of an error later in the macro.

- You can change login IDs after you've accessed a remote database, if TRANSACTION_SCOPE=SINGLE . The SQL language environment disconnects the existing connection, does a commit, and reestablishes the connection using the new user ID and password.
- Do not change login IDs after you have accessed a remote database if TRANSACTION_SCOPE=MULTIPLE, which is the default. The SQL language environment automatically rolls back and a SQL_CODE of -752 is returned, which indicates that the connection could not be changed.

Stored Procedures

A stored procedure is a compiled program stored in a database that can execute SQL statements. In Net.Data, stored procedures are called from Net.Data functions using a CALL statement. Stored procedure parameters are passed in from the Net.Data function parameter list. You can use stored procedures to improve performance and integrity by keeping compiled SQL statements with the database server. Net.Data supports the use of stored procedures with DB2 through the SQL and ODBC language environments. Oracle stored procedures are supported through the Oracle language environment. For DB2 in particular, Net.Data supports stored procedures returning one or more result sets.

This section describes following topics:

- "Stored Procedure Syntax"
- "Calling a Stored Procedure" on page 124
- "Passing Parameters" on page 125
- For DB2 only:"Processing Result Sets from DB2 Stored Procedures" on page 125

Stored Procedure Syntax

The syntax used for stored procedures includes the FUNCTION statement, the CALL statement, and optionally a REPORT block.

```
%FUNCTION  function_name ([IN datatype arg1, INOUT datatype arg2,
    OUT resultsetname, ...]) {
  CALL stored_procedure
[%REPORT [(resultsetname)] { %}]
...
```

```
[%REPORT [(resultsetname)] { %}]
[%MESSAGE %}]

%}
```

Where:

function_name
Is the name of the Net.Data function that initiates the call of the stored procedure

stored_procedure
Is the name of the stored procedure

datatype
Is one of the database data types supported by Net.Data as shown in and Table 7. The data types specified in the parameter list must match the data types in the stored procedure. See your database documentation for more information about these data types.

For DB2 only: *tablename*
Is the name of the Net.Data table in which the result set is to be stored (used only when the result set is to be stored in a Net.Data table. If specified, this parameter name must match the associated parameter name for *resultsetname*.

For DB2 only: *resultsetname*
Is the name that associates a result set returned from a stored procedure with a REPORT block and a table name on the function parm list, or both. The *resultsetname* on a REPORT block must match a *tablename* on the function parameter list.

Table 7. Supported Stored Procedure Data Types

CHAR	FLOAT	TIME
DATE	GRAPHIC	TIMESTAMP
DECIMAL	INTEGER	VARCHAR
DOUBLE	REAL	VARGRAPHIC
DOUBLEPRECISION	SMALLINT	

Important: When Net.Data on Windows or Unix calls a stored procedure in DB2 on OS/390 and OS/400, the stored procedure on these operating systems must use the host variable type DOUBLE or FLOAT when retrieving DECIMAL data from the DB2 database. Using the host variable type DOUBLE or FLOAT will ensure that the returned data is in readable format.

Calling a Stored Procedure

1. Define a function that initiates a call to the stored procedure.

   ```
   %FUNCTION (DTW_SQL) function_name()
   ```

2. (Optional) Specify any IN, INOUT, or OUT parameters for the stored procedure including the result set name of any result sets that are returned from the stored procedure.

   ```
   %FUNCTION (DTW_SQL) function_name (IN datatype
   arg1, INOUT datatype arg2,
       OUT resultsetname...)
   ```

3. Use the CALL statement to identify the stored procedure name.

   ```
   CALL stored_procedure
   ```

4. For DB2: If the DB2 stored procedure is going to generate one result set, optionally specify a REPORT block to define how Net.Data displays the result set.

   ```
   %REPORT  {
   ...
   %}
   ```

   Example:

   ```
   %FUNCTION (DTW_SQL) mystoredproc (IN CHAR(30) arg1 OUT mytable)  {
        CALL myproc
    %REPORT {
     ...
     %ROW {  ...   %}
     ...
    %}
   %}
   ```

5. If the stored procedure is going to generate more than one result set:

   - Specify the result sets as OUT parameters in the FUNCTION statement. The result sets are saved as local tables.

     ```
     %FUNCTION (DTW_SQL) function_name (OUT tablename, ...)
     ```

   - Optionally specify one or more REPORT blocks to define how Net.Data displays the result sets.

     ```
     %REPORT[(resultsetname1)] {
     ...
     %}
     ```

     Example:

     ```
     %FUNCTION (DTW_SQL) mystoredproc (IN CHAR(30) arg1, OUT table1, table2) {
        CALL myproc
      %REPORT (table1) {
           ...
           %ROW {  ...   %}
           ...
      %}
      %REPORT (table1) {
           ...
     ```

```
        %ROW {  ...   %}
        ...
    %}
%}
```

Passing Parameters

You can pass parameters to a stored procedure and you can have the stored procedure update the parameter values so that the new value is passed back to the Net.Data macro. The number and type of the parameters on the function parameter list must match the number and type defined for the stored procedure. For example, if a parameter on the parameter list defined for the stored procedure is INOUT, then the corresponding parameter on the function parameter list must be INOUT. If a parameter on the list defined for the stored procedure is of type CHAR(30), then the corresponding parameter on the function parameter list must also be CHAR(30).

Example 1: Passing a parameter value to the stored procedure

```
%FUNCTION (DTW_SQL) mystoredproc (IN CHAR(30) valuein)  {
    CALL myproc
...
```

Example 2: Returning a value from a stored procedure

```
%FUNCTION (DTW_SQL) mystoredproc (OUT VARCHAR(9) retvalue)  {
  CALL myproc
...
```

Processing Result Sets from DB2 Stored Procedures

You can return one or more result sets from a stored procedure. The result sets can be stored in Net.Data tables for further processing within your macro or processed using a REPORT block. If a stored procedure generates multiple result sets, you must associate a name with each result set generated by the stored procedure. This is done by specifying parameters on the FUNCTION statement. The name you specify for a result set can then be associated with a REPORT block or a Net.Data table, enabling you to determine how each result set is processed by Net.Data. You can:

- Have the result processed in Net.Data's default report style by not defining a report block for the result set.
- Associate a result set with a REPORT block to apply your own report style. In the REPORT block, you can use Net.Data variables, text processing statements like HTML or JavaScript, or other functions to specify how the report data is displayed in the browser.

Result sets are always stored in local tables so that another function in the macro can also access the data. For example, you can pass a Net.Data table to another function so that it can use the data for calculations and display the results based on those calculations.

See “Guidelines and Restrictions for Multiple REPORT Blocks” on page 93 for guidelines and restrictions when using multiple report blocks.

To return a single result set and use default reporting:

Use the following syntax:

```
%FUNCTION (DTW_SQL) function_name (OUT tablename) {
    CALL stored_procedure
%}
```

For example:

```
 %FUNCTION (DTW_SQL) mystoredproc(OUT mytable1) {
    CALL myproc
%}
```

To return a single result set and specify a REPORT block:

Use the following syntax:

```
%FUNCTION (DTW_SQL) function_name (OUT tablename) {
     CALL stored_procedure  [(resultsetname)]
 %REPORT [(resultsetname)] {
  ...
 %}
%}
```

Example1:

```
%FUNCTION (DTW_SQL) mystoredproc (OUT mytable1) {
     CALL  myproc
 %REPORT {
      ...
      %ROW {  ...   %}
      ...
 %}
%}
```

Example 2:

```
%FUNCTION (DTW_SQL) mystoredproc () {
    CALL myproc
 %REPORT  {
  ...
  %ROW {  ...   %}
  ...
 %}
%}
```

To return multiple result sets and display them using default report formatting:

Use the following syntax:

```
%FUNCTION (DTW_SQL) function_name (OUT tablename1, tablename2) {
    CALL stored_procedure
%}
```

Where no report block is specified.

For example:

```
%DEFINE DTW_DEFAULT_REPORT = "YES"
%FUNCTION (DTW_SQL) mystoredproc (OUT mytable1, mytable2) {
    CALL myproc
%}
```

To return multiple result sets and specify REPORT blocks for display processing:

Each result set is associated with its one or more REPORT blocks. Use the following syntax:

```
%FUNCTION (DTW_SQL) function_name (OUT tablename1, tablename2, ...) {
   CALL stored_procedure
 %REPORT (resultsetname1)
   ...
   %ROW {  ...   %}
   ...
 %}
 %REPORT (resultsetname2)
   ...
   %ROW {  ...   %}
   ...
 %}

...
%}
```

For example:

```
%FUNCTION (DTW_SQL) mystoredproc (OUT mytable1, mytable2) {
   CALL myproc

 %REPORT(mytable1) {
  ...
  %ROW {  ...   %}
  ...
 %}

 %REPORT(mytable2) {
  ...
  %ROW {  ...   %}
  ...
 %}
%}
```

Example 2:

```
%FUNCTION(DTW_SQL) mystoredproc(OUT mytable4, OUT mytable3) {
    CALL myproc (mytable1, mytable2, mytable3, mytable4)
    %REPORT(mytable2) {
    ...
    %ROW { ... %}
    ...
    %}
    %REPORT(mytable1) {
    ...
    %ROW { ... %}
    ...
    %}
    %REPORT(mytable4) {
    ...
    %ROW { ... %}
    ...
    %}
 %}
```

The result sets *mytable2*, *mytable1,* and *mytable4* are processed by their corresponding REPORT blocks, in that order, and are displayed as specified. The result sets *mytable4* and *mytable3* are stored into table variables for further processing. The result set *mytable3* will also be displayed using Net.Data's default report format after the three REPORT blocks are done processing.

SQL Language Environment Example

The following example shows a macro with a DTW_SQL function definition that calls an SQL stored procedure. It has three parameters of different data types. The DTW_SQL language environment converts the character string values in each parameter to the correct internal format and passes each parameter by reference to the SQL stored procedure. When the SQL stored procedure completes processing, the updated internal representation is converted to a character string and placed in the corresponding parameter.

```
%{************************************************************
 ************************************************************%}
DEFINE  {
 MACRO_NAME     = "TEST ALL TYPES"
 DTW_HTML_TABLE = "YES"
 Procedure      = "NDLIB.TESTTYPE"
 parm1          = "1"              %{SMALLINT        %}
 parm2          = "11"             %{INT             %}
 parm3          = "1.1"            %{DECIMAL (2,1)   %}
 %}

 %FUNCTION(DTW_SQL) CRTPROC(){
  CREATE PROCEDURE $(Procedure)
  ( INOUT SMALLINT,
  INOUT INT,
  INOUT DECIMAL(2,1))
  EXTERNAL NAME $(Procedure) LANGUAGE C SIMPLE CALL
  %MESSAGE{
    default : "$(DTW_DEFAULT_MESSAGE) : continuing.<br />": continue
```

```
  %}
 %}

 %FUNCTION(DTW_SQL)   myProc
  (INOUT SMALLINT       parm1,
   INOUT INT            parm2,
   INOUT DECIMAL(2,1)   parm3){
CALL $(Procedure)
%}


%HTML(REPORT) {
<head>
<title>Net.Data : SQL Stored Procedure: Example '$(MACRO_NAME)'. <?TITLE>
</head>
<body bgcolor="#bbffff" text="#000000" link="#000000">
<p><p>
Calling the function to create the stored procedure.
<p><p>
 @CRTPROC()
<hr>
<h2>
Values of the INOUT parameters
 prior to calling the stored procedure:<p>
</h2>
<b>parm1  (SMALLINT)</b><br />
$(parm1)<p>
<b>parm2  (INT)</b><br />
$(parm2)<p>
<b>parm3  (DECIMAL)</b><br />
$(parm3)<p>
<p>
<hr>
<h2>
Calling the function that executes the stored procedure.
</h2>
<p><p>
 @myProc(parm1,parm2,parm3)
<hr>
<h2>
Values of the INOUT parameters after
calling the stored procedure:<p>
</h2>
<b>parm1  (SMALLINT)</b><br />
$(parm1)<p>
<b>parm2  (INT)</b><br />
$(parm2)<p>
<b>parm3  (DECIMAL)</b><br />
$(parm3)<p>
</body>
%}
```

System Language Environment

The System language environment supports executing commands and calling external programs.

Issuing Commands and Calling Programs

To issue a command, define a function that uses the System (DTW_SYSTEM) language environment that includes a path to the command to be issued in an EXEC statement. For example:

```
%FUNCTION(DTW_SYSTEM) sys1() {
      %EXEC { /QSYS.LIB/ADDLIBLE.CMD LIB(MYLIBRARY) %}
 %}
```

You can shorten the path to executable objects if you use the EXEC_PATH configuration variable to define paths to directories that contain the objects (such as, commands and programs). See “EXEC_PATH” on page 21 to learn how to define the EXEC_PATH configuration variable.

Example 1: Calls a program

```
%FUNCTION(DTW_SYSTEM) sys3() {
      %EXEC { /QSYS.LIB/MYLIB.LIB/MYPGM.PGM %}
 %
```

Tip: When calling programs, use the Direct Call language environment because it is more efficient and easier to use.

Passing Parameters to Programs

There are two ways to pass information to a program that is invoked by the System (DTW_SYSTEM) language environment, directly and indirectly.

Directly

Pass parameters directly on the call to the program. For example:

```
%DEFINE INPARM1 = "SWITCH1"

%FUNCTION(DTW_SYSTEM) sys1() {
 %EXEC{
  /QSYS.LIB/NETDATA.LIB/RPGCALL1.PGM  ('$(INPARM1)'  'literalstring')
 %}
%}
```

The Net.Data variable INPARM1 is referenced and passed to the program. The parameters are passed to the program in the same way the parameters are passed to the program when the program is called from the command line. The parameters that are passed to the program using this method are considered input parameters, and any modification to the values are not reflected back to Net.Data (the

parameters passed to the program can be used and manipulated by the program, but changes to the parameters are not reflected back to Net.Data).

Indirectly

Pass parameters indirectly, by using *environment variables*. Environment variables are character strings of the form `"name=value"` that are stored in an environment space outside of the program. The strings are stored in a temporary space associated with the process.

When Net.Data calls a DTW_SYSTEM language environment function, the language environment stores any function parameters that are input (IN) or input/output (INOUT) in the environment space prior to executing the statement within the %EXEC block. After the successful completion of the statement, the DTW_SYSTEM language environment determines whether there are any output (OUT or INOUT) function parameters. If so, the language environment retrieves the value corresponding to the function parameter from the environment space and updates the function parameter value with the new value. When Net.Data gets control, it in turn updates all OUT or INOUT parameters with the new values obtained from the DTW_SYSTEM language environment.

Set and retrieve environment variables using the APIs described in Table 8:

Table 8. Environment Variable APIs

ILE Programming Language	To retrieve, use...	To set, use...
C, C++	getenv()	putenv()
CL(1), RPG, COBOL	QtmhGetEnv()(2)	QtmhPutEnv()(3)

1. For OS/400 V3R7 and on, you can also use the CHGENVVAR and ADDENVVAR CL commands to set an environment variable.
2. QtmhGetEnv() is shipped as part of IBM TCP/IP Connectivity Utilities/400.
3. QtmhPutEnv() was not originally shipped as part of IBM TCP/IP ConnectivityUtilities/400 for V3R2 and V3R7. It was added later in the cycle and can be obtained via the V3R2 PTF 5763TC1-SF40953 or the V3R7 PTF 5716TC1-SF40954.

You can pass Net.Data tables to a program called by the System language environment. The program accesses the values of a Net.Data macro table parameter by their Net.Data name. The column headings and field values are contained in variables identified with the table name and column number. For example, in the table `myTable`, the column headings are `myTable_N_`*`j`*, and the field values are

myTable_V_*i*_*j*, where *i* is the row number and *j* is the column number. The number of rows and columns for the table are myTable_*ROWS* and myTable_*COLS*.

System Language Environment Example

The following example shows a macro that uses the System language environment to issue the Send Break Message (SNDBRKMSG) command to all workstation message queues. The text of the message to be sent is constucted from form data (msgToSend).

```
%FUNCTION(DTW_SYSTEM) sndbrkmsg () {
  %EXEC { /QSYS.LIB/SNDBRKMSG.CMD MSG('$(msgToSend)') TOMSGQ(*ALLWS) %}
%}
%HTML(sndbrkmsg) {
 @sndbrkmsg()
%}
```

Chapter 7. Transaction Management with Persistent Macros

Net.Data provides support for transaction processing with persistent macros. A *persistent macro* is a macro that contains built-in functions that enable the macro to run as part of a persistent CGI process in the Web server. This means that multiple blocks of a macro, or multiple macros, can run as part of a single logical transaction.

With non-persistent macros, Net.Data treats each macro invocation as one complete transaction. This means that after each response is sent to the browser, databases are committed, resources are released, and everything is set to an initial state. The next invocation of the same macro results in re-establishing the state of the application based on information passed into the macro as form data or information in the macro itself. There is no capability to save macro variables across invocations, to rollback database changes without explicitly undoing the changes made, or to treat database changes across multiple browser sessions as one complete transaction.

With persistent macros, as an application developer, you can build your application at a transaction level, invoking one or more macros while maintaining a persistent connection. This means variable data is persistent across invocations, so that you no longer need to pass information (such as user login ID) between macro invocations as hidden variables. This includes Net.Data table variables, which cannot be passed across invocations in non-persistent macros. Most important, the application can rollback all the work if the user decides to cancel out while in the middle of a transaction.

See “Invoking a Persistent Macro” on page 48 to learn about invoking persistent macros.

This chapter describes the following topics:

About Persistent Macros

When using persistent macros, Net.Data runs in a special persistent CGI process of the Web server, receives input through standard input and environment variables, and provides data through standard output. However, after the output is returned to the Web server, The Web server does not have

© Copyright IBM Corp. 1997, 2001

to terminate the Net.Data process. Instead, the process remains active, waiting for a response from the user through the Web browser. Because the process does not terminate, Net.Data can maintain state information for the macro and can leave transactions open.

Net.Data communicates to the Web server that it wants to run in a persistent CGI process by sending the server a new HTTP header. Support for the new header, "Accept-HTSession", has been added to the AS/400 HTTP Server in version 4, release 3 (V4R3). Net.Data decides what HTTP headers to send to the server when it sends its first output, because the headers must precede the output. This has the following implications to you when developing a persistent macro:

- Net.Data must know at the time the first output is generated from the macro whether or not this is to be a persistent macro.
- Using the new persistent macro built-in functions, you must specify the macro is persistent before any output is generated.

These restrictions will be noted in the documentation that follows.

The characteristics of persistent Net.Data processes are very similar to those of standard Net.Data processes with the following exceptions:

- They run in a pseudo-connection-oriented environment. The connection between Net.Data and the Web server is persistent, but the connection between the browser and the Web server still has no connection.
- They can have long running transactions. Because a single Net.Data process can span multiple browser requests, transactions can be left open and committed or rolled-back as appropriate based on subsequent browser requests or error conditions.
- A persistent Net.Data process can consume more system resources because it can remain active for potentially long periods of time. Care must be taken in the management of those resources.
- Portability is reduced because the Web server must contain support for persistence.

Defining a Transaction

A transaction can span one HTML block, multiple HTML blocks, or multiple macros. When you specify that you want the macro to be persistent within a transaction, you need to define the start and end of the transaction, as well as which HTML blocks are included in the transaction. Net.Data provides built-in functions that help you complete the following persistent macro tasks:

- "Starting a Transaction" on page 135
- "Specifying the Macro HTML blocks in a Transaction" on page 136
- "Ending a Transaction" on page 139

Starting a Transaction

You start a transaction by indicating to Net.Data that a macro is persistent in your macro before any output is sent to the browser. Net.Data then sends a special HTTP header to the Web server to tell it that the macro requires persistent CGI support.

To start a transaction:

Use one of the following methods in the macro before any output is sent to the Web browser:

- Call the DTW_STATIC() built-in function.

  The DTW_STATIC() function tells Net.Data that the current macro is persistent.

  Syntax: `@DTW_STATIC (["timeout"])`

  Where *timeout* is an optional parameter that specifies the number of seconds the Web server should wait for a response from the browser before ending the transaction.

  Example:

```
@DTW_STATIC("60")
%DEFINE {
  var1 = "val1"
  var2 = "val2"
  %}
...

%HTML(input){
  ...
%}

%HTML(report){
  ...
%}
```

  A timeout value of 60 seconds is specified for this transaction. If a response is not received within 60 seconds from the browser, the Web server ends the transaction. This does not affect the current page on the browser. However, the next page, which would have been part of the transaction, is now part of a new transaction.

- Define a variable with the STATIC attribute.

  Syntax: `%DEFINE(STATIC) var1 = "val1"`

  Example:

```
%DEFINE(STATIC) var1 = "val1"
%DEFINE var2 = "val2"
...
%HTML(input){
...
%}
%HTML(report){
...
%}
```

A statically defined variable keeps its value throughout a transaction, which can span multiple Net.Data invocations.

Specifying the Macro HTML blocks in a Transaction

You define which HTML blocks are a part of your transaction by using an identifier, called the *transaction handle*, in the URL request that invokes the HTML blocks. There are three steps in defining and using a transaction handle:

1. Define the transaction handle in your macro.
2. Call the DTW_ACCEPT built-in function to pass the handle name to Net.Data and the Web server.
3. Specify the handle in the URL request to invoke your next HTML block.

To define a transaction handle:

1. Define a variable for the transaction handle in the DEFINE section. For example:

   ```
   %DEFINE handle=""
   ```

2. Optionally generate a unique transaction handle by specifying the DTW_RTVHANDLE() built-in function in the DEFINE section.

   Syntax: `@DTW_RTVHANDLE(`*`handle_name`*`)`

   Example:

   ```
   @DTW_STATIC()

   %DEFINE handle = ""
   @DTW_RTVHANDLE(handle)
   ```

The transaction handle can be any valid character string. However, the DTW_RTVHANDLE() function provides a measure of security by generating a unique transaction handle, preventing others from invoking a macro which would run in your transaction.

To specify a transaction handle to Net.Data:

Specify the value of the transaction handle to Net.Data with the DTW_ACCEPT() built-in function. Because this handle is part of the information contained in the HTTP headers sent to the server, the

DTW_ACCEPT() function must be called before any output is generated by the macro. Typically, it will be the first element in your HTML block.

Syntax: `@DTW_ACCEPT(`*`handle_name`*`, ["`*`timeout`*`"])`

Where *timeout* is an optional parameter that specifies the number of seconds the Web server should wait for a response from the browser before ending the transaction.

You can call DTW_ACCEPT() within an HTML block or outside of any HTML block. If the function is called outside of any HTML block, the transaction handle and the optional timeout values apply to all HTML blocks within the macro.

Example 1: Specifies a transaction handle for subsequent URL requests to run in this transaction

```
@DTW_STATIC()

%DEFINE handle = ""
@DTW_RTVHANDLE(handle)


%HTML(Block1){
@DTW_ACCEPT(handle)
 ...
%}
```

Important: When you call DTW_ACCEPT() as the first element in the HTML block, ensure that there is no white space between the line on which the %HTML statement is specified and the DTW_ACCEPT() call itself. Net.Data considers the white space as text to send to the browser, and issues an error because the DTW_ACCEPT() call is not found before data is sent to the browser.

Example 2: Specifies a transaction handle which applies to all HTML blocks in the macro

```
@DTW_STATIC()

%DEFINE handle = ""
@DTW_RTVHANDLE(handle)

@DTW_ACCEPT(handle)

%HTML(Block1){
 ...
%}

%HTML(Block2){
 ...
%}
```

To specify the handle when invoking an HTML block that is to run in your transaction:

After you have generated a transaction handle and called the DTW_ACCEPT() function, only URLs with that transaction handle can run in your transaction. The transaction handle must immediately follow the CGI program name in the URL.

Note: When entering the statements in your code, the URL should be on one line with no spaces, but it is split on two lines here for display purposes.

- HTML link:

```
<a href="http://server/Net.Data_invocation_path/transaction_handle/
filename/block/[?name=val&...]">any text</a>
```

- HTML form:

```
<form method=method
 ACTION="http://server/Net.Data_invocation_path/transaction_handle/
filename/block/[?name=val&...]">any text</form>
```

- URL:

```
http://server/Net.Data_invocation_path/transaction_handle/
filename/block/[?name=val&...]
```

Parameters:

server Specifies the name of the Web server. If the server is the local server, you can omit the server name and use a relative URL.

Net.Data_invocation_path
The path and filename of the Net.Data executable file. For example, /cgi-bin/db2www/.

transaction_handle
Specifies which URLs are part of a transaction initiated by a Net.Data macro. The identifier is obtained by calling the DTW_RTVHANDLE built-in function and must follow the *Net.Data_invocation_path.*

filename
Specifies the name of the Net.Data macro. Net.Data searches for and tries to match this file name with the path statements defined in the MACRO_PATH initialization path variable. See “MACRO_PATH” on page 23 for more information.

block Specifies the name of the HTML block in the referenced Net.Data macro.

method Specifies the HTML method used with the form. METHOD=POST is recommended.

?*name***=***val***&...**
Specifies one or more optional parameters passed to Net.Data.

Typically you will provide HTML links to these URLs or specify the URL on a form action tag in your macro.

Example 1: A typical HTML block with links to other macro invocations that run in the same transaction

```
@DTW_STATIC()
 ...
%define handle = ""
@DTW_RTVHANDLE(handle)

%html(report) {
@DTW_ACCEPT(handle)
 ...
<a href="/cgi-bin/db2www/$(handle)/qsys.lib/mylib.lib/
   macros.file/pcgi1.mbr/report2">continue</a><br />
<a href="/cgi-bin/db2www/$(handle)/qsys.lib/mylib.lib/
   macros.file/pcgi1.mbr/quit">quit</a><br />
%}
```

Example 2: A typical HTML block with a FORM ACTION link to another macro

```
@DTW_STATIC()
 ...
%define handle = ""
@DTW_RTVHANDLE(handle)

%html(input) {
@DTW_ACCEPT(handle)
 ...
<form method=post action="/cgi-bin/db2www/$(handle)/qsys.lib/
mylib.lib/macros.file/pcgi1.mbr/report2">
<p>What type of hardware do you want to see?
<menu>
<li><input type="radio" name="hdware" value="MON" checked />Monitors
<li><input type="radio" name="hdware" value="PNT" />Pointing devices
<li><input type="radio" name="hdware" value="PRT" />Printers
<li><input type="radio" name="hdware" value="SCN" />Scanners
</menu>
</form>
%}
```

Ending a Transaction

You end a transaction by indicating to Net.Data that you no longer want your macro to be persistent.

To end the transaction:

Use the DTW_TERMINATE() built-in function to specify the end of a transaction. Like the DTW_ACCEPT() function, this function must be called before any output is generated by the macro and is typically specified as the first element in an HTML block. DTW_TERMINATE tells Net.Data that this invocation is the last invocation in the current transaction.

Syntax: `@DTW_TERMINATE()`

This function does not accept any parameters.

Example:

```
%html(quit) {
@DTW_TERMINATE()
 ...
%}
```

Defining the Scope of a Variable in a Transaction

You can decide what scope you want a variable to have in a transaction by specifying the scope as an attribute of the %DEFINE statement. You can specify

transaction scope
The variable scope is for the entire transaction.

single invocation scope
The variable scope is for a single Net.Data invocation.

To specify transaction scope for a variable:

Specify the attribute STATIC to indicate that the variable has transaction scope, meaning the value of the variable is saved across all invocations in a transaction. STATIC is the default for persistent macros. For example:

```
@dtw_static()
%define(static) var1 = "val1"
```

To specify single invocation scope for a variable:

Specify the attribute TRANSIENT to indicate that the variable has single invocation scope, meaning the value of the variable will be re-initialized on each invocation. TRANSIENT is the default for non-persistent macros. For example:

```
@dtw_static()
%define(transient) var1 = "val1"
```

In a persistent macro:

- All variables that *follow* the DTW_STATIC() call are STATIC if they are not explicitly defined as TRANSIENT.

- All variables that *precede* the DTW_STATIC() call are TRANSIENT if they are not explicitly defined as STATIC.

Specifying COMMIT and ROLLBACK in a Transaction

In a non-persistent macro, a commit or rollback is done implicitly by Net.Data at the end of the macro invocation based on the success or failure of the invocation. With persistent macros, the commit or rollback is now done at the end of the transaction. However, because a transaction can span many macro invocations, you might want to commit or rollback changes incrementally within the transaction.

To commit pending changes during a transaction:

Specify the DTW_COMMIT() built-in function.

This function does not take any parameters and executes all changes pending in the transaction.

For example:

```
%html (report) {
@dtw_accept(handle)
...
%IF (action="Enter")
 @dtw_commit()
%ENDIF

%}
```

To rollback pending changes in the transaction:

Specify the DTW_ROLLBACK() built-in function.

This function does not take any parameters and backs out all changes pending in the transaction.

For example:

```
%html (report) {
@dtw_accept(handle)
...
%IF (action="Cancel")
 @dtw_rollback()
%ENDIF

%}
```

Example of a Persistent Macro

The following simple macro contains multiple HTML blocks that run in a single transaction:

```
@dtw_static()
%define a = "0"
%define(transient) b = "0"
%define handle = ""
@dtw_rtvhandle(handle)

%html(report) {
@dtw_accept(handle)
a = $(a)<br />
b = $(b)<br />
@dtw_add(a, "2", a)
@dtw_add(b, "2", b)
<a href="/cgi-bin/db2www/$(handle)/qsys.lib/mylib.lib/macros.file/pcgi1.mbr/report2">
click here to continue</a><br />
<a href="/cgi-bin/db2www/$(handle)/qsys.lib/mylib.lib/macros.file/pcgi1.mbr/quit">
click here to quit</a><br />
%}

%html(report2) {
@dtw_accept(handle)
a = $(a)<br />
b = $(b)<br />
@dtw_add(a, "2", a)
@dtw_add(b, "2", b)
<a href="/cgi-bin/db2www/$(handle)/qsys.lib/mylib.lib/macros.file/pcgi1.mbr/report3">
Click here to continue</a><br />
<a href="/cgi-bin/db2www/$(handle)/qsys.lib/mylib.lib/macros.file/pcgi1.mbr/quit">
Click here to quit</a><br />
%}

%html(report3) {
@dtw_accept(handle)
a = $(a)<br />
b = $(b)<br />
@dtw_add(a, "2", a)
@dtw_add(b, "2", b)
<a href="/cgi-bin/db2www/$(handle)/qsys.lib/mylib.lib/macros.file/pcgi1.mbr/quit">
Click here to quit</a><br />
%}

%html(quit) {
@dtw_terminate()
a = $(a)<br />
b = $(b)<br />
done
%}
```

Assuming the first call is to the HTML block report, Net.Data:

1. Calls the DTW_STATIC() function, which indicates that this macro is persistent.
2. Creates variable a as a STATIC variable because the default for persistent macros is STATIC.
3. Creates variable b as a TRANSIENT variable because it is explicitly defined with the TRANSIENT attribute.
4. Calls DTW_RTVHANDLE(), which generates a transaction handle and puts it in the variable handle.
5. Starts processing the HTML block report and calls DTW_ACCEPT(), which tells Net.Data what the transaction handle is for this transaction.
6. Finds output to send to the browser, which causes Net.Data to send the HTTP header to the Web server indicating a transaction is starting.
7. Displays the HTML page. The variables a and b both have a value of 0.

After the first page output is sent to the browser, users can choose to either continue with the transaction or quit. If they choose to continue, the Web server invokes URL:

```
/cgi-bin/db2www/$(handle)/qsys.lib/mylib.lib/macros.file/pcgi1.mbr/report2
```

The Web server recognizes the transaction handle as the one specified by Net.Data in the HTTP header. It invokes Net.Data as a persistent CGI program, which means the macro invocation is part of the current transaction.

When the HTML block report2 is invoked, Net.Data:

1. Calls the DTW_STATIC() function, which indicates this macro is persistent.
2. Recognizes that variable a is a STATIC variable and keeps the current value rather than re-initializing it to 0.
3. Recognizes that variable b is a TRANSIENT variable, creates a new instance of the variable, and initializes it to 0.
4. Calls DTW_RTVHANDLE(), which generates a transaction handle and puts it in the variable handle.
5. Starts processing the HTML block report2 and calls DTW_ACCEPT(), which tells Net.Data what the transaction handle is for this transaction.
6. Finds output to send to the browser, which causes Net.Data to send the HTTP header to the server indicating a transaction is continuing.
7. Displays the HTML page. Variable *a* will have a value of 2 and variable b will have a value of 0. The value of variable a is saved from the previous invocation because it is a static variable. The value of variable b is reset to 0.

After the second page is sent to the browser, the user can choose to either continue with the transaction or quit. If they choose to quit, the Web server invokes the following URL:

```
/cgi-bin/db2www/$(handle)/qsys.lib/mylib.lib/macros.file/pcgi1.mbr/quit
```

The Web server recognizes the transaction handle as the one specified by Net.Data in the HTTP header, and invokes Net.Data as a persistent CGI program, which means the macro invocation is part of the current transaction.

When the HTML block `quit` is invoked, Net.Data:

1. Calls the DTW_STATIC() function, which indicates this macro is persistent.
2. Recognizes that variable `a` is a STATIC variable and keeps the current value rather than re-initializing it to 0.
3. Recognizes that variable `b` is a TRANSIENT variable, creates a new instance of the variable, and initializes it to 0.
4. Calls DTW_RTVHANDLE(), which generates a transaction handle and puts it in the variable handle.
5. Starts processing the HTML block `quit` and calls DTW_TERMINATE(), which tells Net.Data that this is the last invocation in this transaction.
6. Finds output to send to the browser, which causes Net.Data to send the HTTP header to the server indicating a transaction is ending.
7. Displays the HTML page. Variable `a` has a value of 4 and variable `b` has a value of 0.
8. Cleans up all variables and other resources that have transaction level scope, because the DTW_TERMINATE() call has been executed.

Chapter 8. Improving Performance

Improving performance is an important part of tuning your system. This chapter discusses strategies for improving the performance of Net.Data. The following topics are discussed:

- Using the web Server APIs
- "Net.Data Caching of Macros"
- "Optimizing the Language Environments"

In addition, ensure that your Web server has been properly tuned. The performance of your Web server has a direct effect on response time, independently of how fast Net.Data processes a macro or direct request.

Net.Data Caching of Macros

With Net.Data for OS/400, macro caching is enabled by default and is used to improve throughput and reduce CPU utilization. When macro caching is enabled, preprocessed macros are cached in memory when the macros are first invoked. These preprocessed versions are then available for reuse, thereby eliminating the costs associated with reading in the macros from HFS and processing them each time they are requested. The cached version of a macro is available to a requestor that has read permission for the file containing the macro.

The amount of memory that the preprocessed version of the macro uses is approximately twice the size of the macro itself. You can control the amount of memory that will be used for the caching of macros by using the caching configuration variable. For more information on using this variable, see "DTW_MACRO_CACHE_SIZE: Macro Cache Size Variable" on page 12.

Optimizing the Language Environments

The following sections describes techniques you can use to improve performance when using the Net.Data-provided language environments.

- "REXX Language Environment"
- "SQL Language Environment" on page 146
- "System Language Environment" on page 147

REXX Language Environment

Use the following tips to improve the performance of your Net.Data application:

© Copyright IBM Corp. 1997, 2001

- Combine your REXX programs where possible. Having fewer, larger programs provides better performance than more smaller programs because the REXX interpreter is initialized each time a REXX language environment function is called in the macro.
- Store the REXX program in an external file instead of including the REXX program inline in the Net.Data macro.
- For external REXX programs, reference the global variables on the command line in the %EXEC statement.
- Pass input-only parameters directly to a REXX program by defining global Net.Data variables and referencing the variables. For inline REXX programs, reference the global variables directly in your REXX source.
- To avoid the overhead of launching the REXX interpreter, consider using MACRO_FUNCTION blocks as an alternative to REXX programs.

SQL Language Environment

To learn about DB2 performance considerations, see *DB2 for 0S/400 SQL Programming*. This publication has a wealth of information, such as effectively using SQL indexes, improving performance of join queries, and improving performance when selecting data from more than two tables.

Use the following SQL language environment techniques to improve performance.

- Reduce the number of user IDs connecting to a database to avoid reconnecting to the database. The SQL language environment associates a user-profile and password to any remote connections to databases that it establishes. If the LOGIN and PASSWORD variables do not match the user-profile and password associated with an opened connection, the connection is closed and re-established, and the LOGIN and PASSWORD values are associated with the re-opened connection.
- Use the START_ROW_NUM and RPT_MAX_ROWS Net.Data variables to reduce the size of returned tables. If, on a SELECT SQL statement, the result set contains hundreds of records, return a subset of the result set back to the browser by using the START_ROW_NUM like a scrollable cursor and RPT_MAX_ROWS to limits the number of records returned. You should be aware that Net.Data reissues the query every time since there is no notion of state. However, you can use Net.Data support for persistent macros to store the result set in a Net.Data table that persists for the life of the transaction. See “Chapter 7. Transaction Management with Persistent Macros” on page 133 to learn more about persistent Net.Data macros.
- When you have SQL statements where the only information that changes is the input values in a WHERE clause, consider taking advantage of the DTW_USE_DB2_PREPARE_CACHE feature of Net.Data. Set this value to ″YES″ in the initialization file, or in individual macros if you do not want it to apply globally. This setting tells Net.Data to use host variables for the

input values, helping DB2 prepare statements more quickly. See (some section in Chapter 6) for more information on how to use this variable.

- Consider using a stored procedure to handle complex database tasks. Using embedded SQL and understanding the structure of result sets reduces the overhead Net.Data uses to dynamically describe results. For more information on the performance trade-offs when using stored procedures, see the *DB2 Administration Guide*.

  Note that starting in OS/400 V4R2, the SQL engine has a prepared statement cache. Using the cache, the SQL engine stores away information about prepared statements, and keeps this information in system-wide storage. Then, when the same statement is executed again, even if its by a different user and a different job, the statement will run much faster. The system-wide prepared statement cache is part of normal SQL processing and requires no user action to configure or enable it. The cache may reduce any performance benefits that the static SQL might have over dynamic SQL.

System Language Environment

Pass input-only parameters directly to the program that the System language environment is invoking by defining global Net.Data variables and referencing the variables.

Net.Data for OS/400 has introduced a new language environment called Direct Call, which provides easier and more efficient interface for calling programs. Use the System language environment to issue commands; use the Direct Call language environment to call programs

Chapter 9. Serviceability Features

The following sections describe tracing and error reporting features for Net.Data.

Net.Data Trace Log

Tracing allows you to monitor Net.Data as it processes a macro. As Net.Data executes a macro, it will write out trace information that includes the functions being called, the parameters being passed (both input and output), and the errors that may be encountered.

Tracing is mainly used for service calls, but it can also be used by customers who are debugging their applications.

To enable tracing, you need to set where the trace log is stored and what level of trace data Net.Data needs to capture. This is done by setting the configuration variables DTW_TRACE_LOG_DIR and DTW_TRACE_LOG_LEVEL, respectively. For more information, see "DTW_TRACE_LOG_LEVEL: Level of Trace to Log" on page 17 and "DTW_TRACE_LOG_DIR: Location of Trace File" on page 17.

Net.Data logs trace records to the file, `netdata.trace`. In order for Net.Data to successfully write trace records to the trace log file, the user IDs under which Net.Data executes must have:

- Write authority on the log directory specified in the DTW_TRACE_LOG_DIR configuration variable.
- Execute authority on all directories in the path, including the log directory.

By default, trace records from the processing of a macro running in one thread are interleaved with trace records from the processing of a macro running in another thread. The default works well if you have total control over what URL is to be invoked against your Web site (as is the case in a development environment). However, in a high traffic environment, having the trace records of the processing of other macros interleaved with trace records that you may be interested in makes it hard to follow the flow of the processing of a macro. You can indicate to Net.Data whether trace records should or should not be merged into one file by setting the value of the configuration variable, DTW_TRACE_MERGE_RECORDS to NO, in which case Net.Data will log trace records to a file named netdata.trace.XXXX, where

© Copyright IBM Corp. 1997, 2001

'XXXX' is the process/thread identifier. For more information, see "DTW_TRACE_MERGE_RECORDS: Merge Trace Records" on page 18.

If tracing is enabled, you can write your own trace messages to the Net.Data trace. To do this, simply pass the message as a parameter to the built-in function DTW_LOG_TRACEMSG(). See the *IBM Net.Data Reference* book for more information on how to use the built-in function DTW_LOG_TRACEMSG().

Net.Data Error Log

Error logging allows you to have Net.Data errors logged to a file. If you are in the process of creating or enhancing a Net.Data application, you may want to log all error messages to ensure that known or expected errors are caught by message blocks. You may also want to track down unexpected errors. In production phase, log all uncaught error messages. Uncaught error messages indicate that something is wrong with your application. Periodically, check the error log to see if your application is working as expected. If error logging is disabled, the only way that you can know if something is wrong with your application is from your users.

To enable error logging, you need to set where the error log is stored and what level of error messages Net.Data needs to capture. This is done by setting the configuration variables DTW_ERROR_LOG_DIR and DTW_ERROR_LOG_LEVEL, respectively. For more information, see "DTW_ERROR_LOG_LEVEL: Level of Error to Log" on page 11 and "DTW_ERROR_LOG_DIR: Location of Error Log" on page 11.

Net.Data logs error records to the file, `netdata.error.log`. In order for Net.Data to successfully write error records to the error log file, the user IDs under which Net.Data executes must have:

- Write authority on the log directory specified in the DTW_ERROR_LOG_DIR configuration variable.
- Execute authority on all directories in the path, including the log directory.

If error logging is enabled, you can write your own error records to the Net.Data error log. To do this, simply pass the message as a parameter to the built-in function DTW_LOG_ERRORMSG(). See the *IBM Net.Data Reference* book for more information on how to use the built-in function DTW_LOG_ERRORMSG().

Appendix A. Bibliography

Net.Data Technical Library

The Net.Data Technical Library is available from the Net.Data Web site at
`http://www.ibm.com/software/data/net.data/library.html`

Document	Description
• *Net.Data Administration and Programming Guide for OS/390* • *Net.Data Administration and Programming Guide for OS/2, Windows NT, and UNIX* • *Net.Data Administration and Programming Guide for OS/400*	Contains conceptual and task information about installing, configuring, and invoking Net.Data. Also describes how to write Net.Data macros, use Net.Data performance techniques, use Net.Data language environments, manage connections, and use Net.Data logging and traces for trouble shooting and performance tuning.
Net.Data Reference	Describes the Net.Data macro language, variables, and built-in functions.
Net.Data Language Environment Interface Reference	Describes the Net.Data language environment interface.
Net.Data Messages and Codes Reference	Lists Net.Data error messages and return codes.

Related Documentation

The following documents might be useful when using Net.Data and related products:

- *DB2 for 0S/400 SQL Programming*
- *OS/400 Distributed Database Programming*

Additionally, OS/400 documentation and redbooks, including books about DB2, are available at the following URL:
`http://publib.boulder.ibm.com/html/as400/infocenter.html`

© Copyright IBM Corp. 1997, 2001

Appendix B. Net.Data Sample Macro

This sample macro application displays a list of employees names from which the application user can obtain additional information about an individual employee by selecting the employee's name from the list. The macro uses the SQL language environment to query the EMPLOYEE table for both the employee names and the information about a specific employee.

The macro uses an include file, which contains the DEFINE block for the macro.

Figure 12 on page 154 shows the sample macro. Figure 13 on page 156 shows the include file.

© Copyright IBM Corp. 1997, 2001

```
%{************************ Sample Macro ****************************
*   FileName = sqlsamp1.dtw                                          *
*   Description:                                                     *
*     This Net.Data macro queries...                          *
*         - The EMPLOYEE table to create a selection list of         *
*          employees for display at a browser                        *
*         - The EMPLOYEE table to obtain additional information      *
*          about an individual employee                              *
*                                                                    *
**********************************************************************%}
%{*****************************************************************************
*   Include for global DEFINEs -                                                *
*****************************************************************************%}
%INCLUDE "sqlsamp1.hti"
%}
%{*****************************************************************************
*  Function:  queryDB              Language Environment: SQL                  *
*  Description: Queries the table designated by the variable myTable and      *
*  creates a selection list from the result.  The value of the variable       *
*  myTable is specified in the include file sqlsamp1.hti.                     *
*****************************************************************************%}
%FUNCTION(DTW_SQL) queryDB() {
 SELECT FIRSTNME FROM EMPLOYEE
%MESSAGE {
    -204:  {<p><b>ERROR -204: Table EMPLOYEE not found. </b> </p>
           %} : exit
    +default: "WARNING $(RETURN_CODE)" : continue
    -default: "Unexpected ERROR $(RETURN_CODE)" : exit
%}

%REPORT {
<select name="emp_name">
%ROW{
<option>$(V1)</option>
%}
</select>
%}
%}

%{*****************************************************************************
*  Function: fname         Language Environment: SQL                          *
*  Description: Queries the table designated by the variable myTable for      *
*               additional information about the employee identified by the   *
*               variable emp_name.                                            *
*****************************************************************************%}
%FUNCTION(DTW_SQL) fname(){
SELECT FIRSTNME, PHONENO, JOB FROM EMPLOYEE WHERE FIRSTNME='$(emp_name)'
%MESSAGE {
  -204: "Error -204: Table not found "
  -104: "Error -104: Syntax error"
   100: "Warning 100: No records" : continue
  +default: "Warning $(RETURN_CODE)" : continue
  -default: "Unexpected SQL error" : exit
%}
%}
```


Figure 12. Sample macro (Part 1 of 3)

```
%{*********************************************************************
*  HTML block: INPUT            Title: Dynamic Query Selection        *
*                                                                     *
*  Description: Queries the EMPLOYEE table to create a selection list *
*               of the employees for display at the browser           *
*********************************************************************%}
%HTML(INPUT) {
<html>
<head>
<title>Generate Employee Selection List</title>
</head>
<body>
<h3>$(exampleTitle)</h3>
<p>This example queries a table and uses the result to create
a selection list using a <em>%REPORT</em> block. </p>
<hr />
<form method="post" action="report">
@queryDB()
<input type="submit" value="Select Employee" />
</form>
<hr />
</body>
</html>
%}
```

Figure 12. Sample macro (Part 2 of 3)

```
%{*****************************************************************************
*  HTML block:    REPORT                                                      *
*  Description: Queries the EMPLOYEE table to obtain additional information *
*               about an individual employee                                 *
*****************************************************************************%}
%HTML(REPORT) {
<html>
<head>
<title>Obtain Employee Information</title>
</head>
<body>
<h3>You selected employee name = $(emp_name)</h3>
<p>Here is the information for that employee:
<pre>
@fname()
</pre></p>
<hr /><a href="input">Return to previous page</a>
</body>
</html>
%}

%{     End of Net.Data macro 1 %}
```

Figure 12. Sample macro (Part 3 of 3)

```
================================================================================
%{*************************** Include File *********************************
*   FileName = sqlsamp1.hti                                                   *
*   Description:                                                              *
*     This include file provides global DEFINEs for the sqlsamp1.dtw          *
*     Net.Data macro.                                                         *
*****************************************************************************%}
%define {
    emp_name   =""
    reposition = sign
    exampleTitle = "Sample Macro"
    %}

%{     End of include file  %}
```

Figure 13. Include file

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

© Copyright IBM Corp. 1997, 2001

be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this publication to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is as your own risk.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

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

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

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

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Trademarks

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

AIX
AS/400
DB2
DB2 Universal Database
DRDA
DataJoiner
IBM
IMS
Language Environment
MVS/ESA
Net.Data
OS/2
OS/390
OS/400
OpenEdition

The following terms are trademarks of other companies as follows:

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

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

Lotus and Domino Go Webserver are trademarks of Lotus Development Corporation in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Index

© Copyright IBM Corp. 1997, 2001

F

G

H

I

J

L

M

N

P

R

S

T

U

V

W

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