IBM Informix SQL

Reference Manual

Version 7.3
January 2002
Part No. 000-5471A

Note:
Before using this information and the product it supports, read the information in the appendix entitled "Notices."

Table of Contents

Chapter 3 The PERFORM Screen Transaction Processor

Chapter 4 The ACE Report Writer

Introduction

In This Introduction

This introduction provides an overview of the information in this manual and describes the conventions it uses.

About This Manual

This manual is a complete reference to the facilities that make up INFORMIX-SQL. It contains information about everything you can do with INFORMIX-SQL, and it is organized by facility name. Once you have used the *INFORMIX-SQL User Guide* and are familiar with INFORMIX-SQL basics, you can use this manual to learn about advanced features and to quickly locate specific information.

Organization of This Manual

The manual includes the following chapters and appendixes:

- Chapter 1, "The INFORMIX-SQL Main Menu," explains how to use the INFORMIX-SQL Main menu and describes what each option on the menu does.
- Chapter 2, "The FORMBUILD Transaction Form Generator," focuses on FORMBUILD, supplying the information needed to build a screen form.
- Chapter 3, "The PERFORM Screen Transaction Processor," considers each PERFORM menu option in detail, explaining how to enter, modify, remove, and retrieve data.
- Chapter 4, "The ACE Report Writer," lists the formatting features you can use with ACE to prepare custom reports.

- Chapter 5, "User-Menu," describes the User-menu option and provides the information needed to build a menu.
- Chapter 6, "Functions in ACE and PERFORM," shows you how to call C functions from ACE reports and PERFORM forms.
- Appendix A, "The Demonstration Database and Examples," describes the sample forms and reports used in this manual and the *INFORMIX-SQL User Guide*.
- Appendix B, "Setting Environment Variables," describes how to use environment variables and documents a few environment variables specific to INFORMIX-SQL. For detailed documentation of environment variables for all Informix products, see the *Informix Guide to SQL: Reference*.
- Appendix C, "Global Language Support," describes Global Language Support (GLS), and how it affects INFORMIX-SQL in non-US-English environments.
- Appendix D, "Modifying termcap and terminfo," discusses how to modify **termcap** and **terminfo** files to use special graphics characters in the FORMBUILD transaction processor.
- Appendix E, "The ASCII Character Set," is an ASCII chart.
- Appendix F, "Reserved Words," lists reserved words for all Informix products.
- Appendix G, "Accessing Programs from the Operating System," demonstrates how to access each INFORMIX-SQL program from the command line.

Types of Readers

This manual is written for all INFORMIX-SQL developers. You do not need database management experience or familiarity with relational database concepts to use this manual. A knowledge of SQL (Structured Query Language), however, and experience using a high-level programming language would be useful.

Software Dependencies

This manual is written with the assumption that you are using an Informix database server, Version 7.x or later.

You can easily use applications developed with an earlier version of INFORMIX-SQL, such as Version 4.x or 6.x or 7.2, with this version of INFORMIX-SQL.

Assumptions About Your Locale

Informix products can support many languages, cultures, and code sets. All culture-specific information is brought together in a single environment, called a GLS (Global Language Support) locale.

The examples in this manual are written with the assumption that you are using the default locale, **en_us.8859-1**. This locale supports U.S. English format conventions for dates, times, and currency. In addition, this locale supports the ISO 8859-1 code set, which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

If you plan to use non-ASCII characters in your data or your SQL identifiers, or if you want to conform to the nondefault collation rules of character data, you need to specify the appropriate nondefault locale.

For instructions on how to specify a nondefault locale, additional syntax, and other considerations related to GLS locales, see the *Informix Guide to GLS Functionality*.

Demonstration Database and Examples

INFORMIX-SQL includes a demonstration database called **stores7** that contains information about a fictitious wholesale sporting-goods distributor. You can create the **stores7** database in any directory you like by changing to the directory and entering the following command:

```
isqldemo
```

Many (but not all) of the examples in the INFORMIX-SQL documentation set are based on the **stores7** database. This database is described in detail in the *Informix Guide to SQL: Reference*. The examples are installed with your software in the **$INFORMIXDIR/demo/sql** directory. For U.S. English, go to the **en_us/0333** subdirectory; for other languages, go to the appropriate subdirectory under the **fgl** directory.

Documentation Conventions

This section describes the conventions that this manual uses. These conventions make it easier to gather information from this and other volumes in the documentation set. The following conventions are discussed:

- Typographical conventions
- Icon conventions
- Example-code conventions
- Syntax conventions

Typographical Conventions

This manual uses the following conventions to introduce new terms, illustrate screen displays, describe command syntax, and so forth.

Convention	Meaning
KEYWORD	All primary elements in a programming language statement (keywords) appear in uppercase letters in a serif font.
italics ***italics*** `italics`	Within text, new terms and emphasized words appear in italics. Within syntax diagrams and code examples, identifiers or values that you are to specify appear in italics.
boldface ***boldface***	Names of program entities (such as classes, events, and tables), environment variables, file and pathnames, and interface elements (such as icons, menu items, and buttons) appear in boldface.
`monospace` *`monospace`*	Information that the product displays and information that you enter appear in a monospace typeface.

Tip: *When you are instructed to "enter" characters or to "execute" a command, immediately press* RETURN *after the entry. When you are instructed to "type" the text or to "press" other keys, no* RETURN *is required.*

Icon Conventions

Throughout the documentation, you will find text that is identified by several different types of icons. This section describes these icons.

Feature, Product, and Platform Icons

Feature, product, and platform icons identify paragraphs that contain feature-specific, product-specific, or platform-specific information.

Icon	Description
GLS	Identifies information that relates to the Informix Global Language Support (GLS) feature
IDS	Identifies information or syntax that is specific to Informix Dynamic Server and its editions
SE	Identifies information or syntax that is specific to INFORMIX-SE

These icons can apply to a row in a table, one or more paragraphs, or an entire section. A ♦ symbol indicates the end of the feature-specific, product-specific, or platform-specific information.

Additional Documentation

For additional information, you might want to refer to the following types of documentation:

- Documentation included with INFORMIX-SQL
- On-line manuals
- On-line error messages
- Related reading

Syntax Conventions

This section describes conventions for syntax diagrams. Each diagram displays the sequences of required and optional keywords, terms, and symbols that are valid in a given statement or segment, as Figure 1 shows.

Figure 1
Example of a Simple Syntax Diagram

Each syntax diagram begins at the upper-left corner and ends at the upper-right corner with a vertical terminator. Between these points, any path that does not stop or reverse direction describes a possible form of the statement. (For a few diagrams, however, notes in the text identify path segments that are mutually exclusive.)

Syntax elements in a path represent terms, keywords, symbols, and segments that can appear in your statement. The path always approaches elements from the left and continues to the right, except in the case of separators in loops. For separators in loops, the path approaches counterclockwise. Unless otherwise noted, at least one blank character separates syntax elements.

Elements That Can Appear on the Path

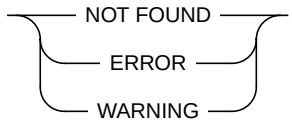
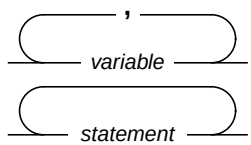
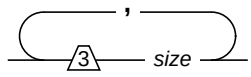
You might encounter one or more of the following elements on a path.

Element	Description
KEYWORD	A word in UPPERCASE letters is a keyword. You must spell the word exactly as shown; you can, however, use either uppercase or lowercase letters.
(. , ; @ + * - /)	Punctuation and other nonalphanumeric characters are literal symbols that you must enter exactly as shown.
" " ' '	Double quotes must be entered as shown. If you prefer, you can replace the pair of double quotes with a pair of single quotes, but you cannot mix double and single quotes.

(1 of 3)

Element	Description
variable	A word in italics represents a value that you must supply. A table immediately following the diagram explains the value.
ATTRIBUTE Clause p. 3-288 ATTRIBUTE Clause	A reference in a box represents a subdiagram. Imagine that the subdiagram is spliced into the main diagram at this point. When a page number is not specified, the subdiagram appears on the same page. The aspect ratios of boxes are not significant.
SELECT Statement *see* SQL:S	A reference to SQL:S in a syntax diagram represents an SQL statement or segment that is described in the *Informix Guide to SQL: Syntax*. Imagine that the segment were spliced into the diagram at this point.
SE	An icon is a warning that this path is valid only for some products, or only under certain conditions. Characters on the icons indicate what products or conditions support the path. These icons appear in some syntax diagrams: SE — This path is valid only for INFORMIX-SE database servers. IDS — This path is valid only for Informix Dynamic Server.
ALL	A shaded option is the default, if you provide no other specification.
...	A syntax segment within a pair of arrows is a subdiagram.
	The vertical line terminates the syntax diagram.
IS NULL / NOT	A branch below the main path indicates an optional path. (Any term on the main path is required, unless a branch can circumvent it.)

(2 of 3)

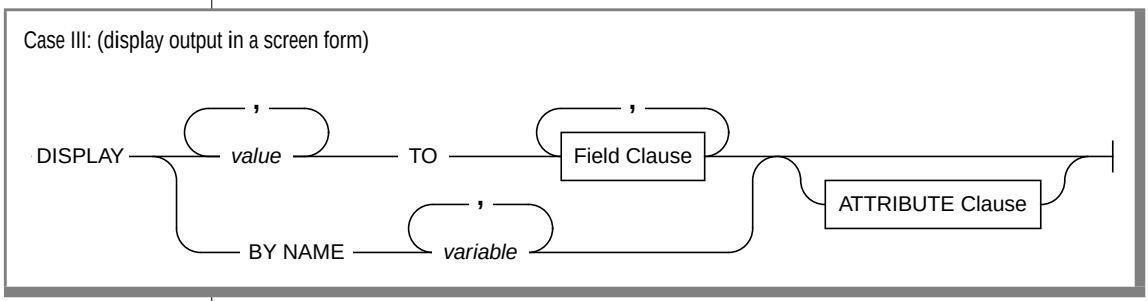
Element	Description
	A set of multiple branches indicates that a choice among more than two different paths is available.
	A loop indicates a path that you can repeat. Punctuation along the top of the loop indicates the separator symbol for list items. If no symbol appears, a blank space is the separator, or (as here) the Linefeed that separates successive statement in a source module.
	A gate (3) on a path indicates that you can only use that path the indicated number of times, even if it is part of a larger loop. You can specify *size* no more than three times within this statement segment.

(3 of 3)

How to Read a Syntax Diagram

Figure 2 shows a syntax diagram that uses some of the path elements that the previous table lists.

Figure 2
Example of a Syntax Diagram



The "Case III" label above the diagram implies that this statement can have at least two other syntax patterns. To use this diagram to construct a statement, start at the top left with the keyword DISPLAY. Then follow the diagram to the right, proceeding through the options that you want.

The following steps are shown in the diagram:

1. Type the keyword DISPLAY.
2. You can display the values of a list of variables to an explicit list of fields within the current screen form:
   - Type the name of a *variable*. If you want to display the values of several variables, separate successive variables by comma.
   - Type the keyword TO after the name of the last variable.
   - Type the name of a *field* in the current form in which to display the first variable. To find the syntax for specifying field names, go to the "Field Clause" segment on the specified page.
3. If you are using a form whose fields have the same names as the variables that you want to display, you can follow the lower path:
   - Type the keywords BY NAME after DISPLAY.
   - Type the name of a *variable*. If you want to display the values of several variables, separate successive variables by comma.
4. You can optionally set a screen attribute for the displayed values:
   - Use the syntax of the "ATTRIBUTE Clause" segment on the specified page to specify the screen attribute that you desire.
5. Follow the diagram to the terminator.

   Your DISPLAY TO or DISPLAY BY NAME statement is now complete.

   A restriction on step 2 (that there must be as many fields as variables) appears in notes that follow the diagram, rather than in the diagram itself. "Usage" notes also follow the syntax for each statement.

Documentation Included with INFORMIX-SQL

The INFORMIX-SQL documentation set includes the following manuals.

Manual	Description
INFORMIX-SQL Reference Manual	A complete reference to the programs that make up INFORMIX-SQL. It contains information about everything you can do with INFORMIX-SQL and is organized by program name. Once you have used the *INFORMIX-SQL User Guide* and are familiar with INFORMIX-SQL basics, you can use the *INFORMIX-SQL Reference Manual* to learn about advanced features and to quickly locate specific information.
INFORMIX-SQL User Guide	Introduces INFORMIX-SQL and provides the context needed to understand the other manuals in the documentation set. You do not need database management experience or familiarity with basic database management concepts to use this manual. It includes general information about database systems and leads you through the steps necessary to create a database, enter and access database information, and produce printed reports.
Informix Guide to SQL: Tutorial	Provides a tutorial on SQL as it is implemented by Informix products, and describes the fundamental ideas and terminology that are used when planning and implementing a relational database. It also describes how to retrieve information from a database, and how to modify a database.
Informix Guide to SQL: Reference	Provides full information on the structure and contents of the demonstration database that is provided with INFORMIX-SQL. It includes details of the Informix system catalog tables, describes Informix and common environment variables that should be set, and describes the column data types that are supported by Informix database engines. It also provides a detailed description of all of the SQL statements that Informix products support.
Informix Guide to SQL: Syntax	Contains syntax diagrams for all of the SQL statements and statement segments that are supported by the 7.3 database server.
Informix Guide to GLS Functionality	Provides full information about using Global Language Support features.
Informix Error Messages	Provides error messages organized by error number. When an error occurs you can look it up by number and learn its cause and solution.

On-Line Manuals

The Informix Answers OnLine CD allows you to print chapters or entire books and perform full-text searches for information in specific books or throughout the documentation set. You can install the documentation or access it directly from the CD. For information about how to install, read, and print on-line manuals, see the installation insert that accompanies Answers OnLine. You can also access Answers OnLine on the Web at the following URL: `www.informix.com/answers`.

Useful On-Line Files

In addition to the Informix set of manuals, the following on-line files, located in the **$INFORMIXDIR/release** directory, may supplement the information in the *INFORMIX-SQL User Guide* and *Reference Manual*:

Documentation Notes describe feature and performance topics not covered in the manual or that have been modified since publication. The file containing the Documentation Notes for INFORMIX-SQL is called **ISQLDOC_7.3**.

Release Notes describe performance differences from earlier versions of Informix products and how these differences may affect current products. The file containing the Release Notes for INFORMIX-SQL and other products is called **TOOLS_7.3**.

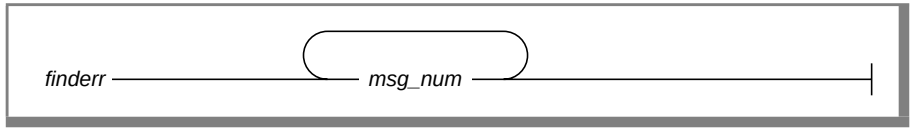
On-Line Help

INFORMIX-SQL provides on-line Help; invoke Help by pressing CONTROL-W.

On-Line Error Messages

Use the **finderr** script to display a particular error message or messages on your screen. The script is located in the **$INFORMIXDIR/bin** directory.

The **finderr** script has the following syntax.



msg_num Indicates the number of the error message to display. Error message numbers range from -1 to -32000. Specifying the - sign is optional.

For example, to display the -359 error message, you can enter either of the following:

```
finderr -359
```

or, equivalently:

```
finderr 359
```

The following example demonstrates how to specify a list of error messages. The example also pipes the output to the UNIX **more** command to control the display. You can also direct the output to another file so that you can save or print the error messages:

```
finderr 233 107 113 134 143 144 154 | more
```

A few messages have positive numbers. These messages are used solely within the application tools. In the unlikely event that you want to display them, you must precede the message number with the + sign.

The messages numbered -1 to -100 can be platform-dependent. If the message text for a message in this range does not apply to your platform, check the operating system documentation for the precise meaning of the message number.

Related Reading

The following Informix database server publications provide additional information about the topics that this manual discusses:

- Informix database servers and the SQL language are described in separate manuals, including *Informix Guide to SQL: Tutorial*, *Informix Guide to SQL: Syntax*, and *Informix Guide to SQL: Reference*.

- Information about setting up Informix database servers is provided in the *Administrator's Guide* for your particular server.
- The *Informix Guide to GLS Functionality* describes how to use Global Language Support to create applications for international markets.

Informix Press, in partnership with Prentice Hall, publishes books about Informix products. Authors include experts from Informix user groups, employees, consultants, and customers. You can access Informix Press on the Web at the following URL: `www.informix.com/ipress`.

Informix Welcomes Your Comments

Let us know what you like or dislike about our manuals. To help us with future versions of our manuals, we want to know about any corrections or clarifications that you would find useful. Include the following information:

- The name and version of the manual that you are using
- Any comments that you have about the manual
- Your name, address, and phone number

Write to us at the following address:

Informix Software, Inc.
SCT Technical Publications Department
4100 Bohannon Drive
Menlo Park, CA 94025

If you prefer to send electronic mail, our address is:

`doc@informix.com`

We appreciate your suggestions.

Important: *The* ***doc*** *alias is monitored only by the Informix departments that create and maintain manuals and on-line documentation files. It is not an appropriate channel for technical support issues, sales inquiries, or questions about the availability of Informix products.*

Chapter

1

The INFORMIX-SQL Main Menu

In This Chapter

This chapter gives an overview of INFORMIX-SQL and details about the main menu.

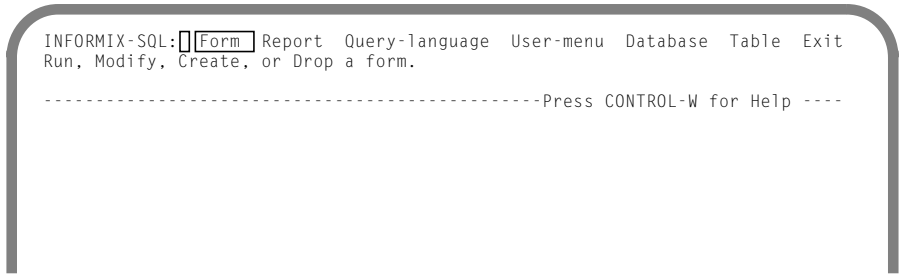
Product Overview

INFORMIX-SQL is a computer-based record-keeping system. As a *database management system*, INFORMIX-SQL consists of useful programs or modules that perform data management tasks. INFORMIX-SQL can substantially reduce the amount of time required to organize, store, and retrieve information. It can summarize, group, and format information in a variety of helpful ways. With INFORMIX-SQL, you can perform these database management tasks:

- Create, modify, and drop databases and tables
- Load data from operating system files
- Run queries using an interactive query language
- Insert, delete, update, and query on data in the database
- Create and drop privileges and indexes
- Create and compile custom forms or reports
- Create and run custom menus

Accessing INFORMIX-SQL

To begin working with INFORMIX-SQL, enter `isql` at the operating-system prompt. At this point, INFORMIX-SQL displays the Main menu.



The INFORMIX-SQL Screens

The INFORMIX-SQL menu system uses two kinds of screens: a menu screen, like the INFORMIX-SQL Main menu, and a text-entry screen.

Menu Screens

The top line of a menu screen lists your options. One option is always highlighted. The second line gives a brief description of the highlighted option. Each time you press the SPACEBAR, the highlight moves to the next option and the description changes. You can also use the [→] and [←] keys to move the highlight. The fourth line displays the name of the current database and the following message:

```
Press CONTROL-W for Help
```

Selecting Options

You can normally select menu options in two ways:

- Use the SPACEBAR to move the highlight over the option you want to choose and press RETURN.
- Type the first letter of the option you want to select. Case is not important—you can type `t` or `T` to select the Table option.

INFORMIX-SQL displays the screen for the menu option you have selected.

Exiting the Menu

Each menu has an Exit option. To leave a menu screen, type `e` for Exit. INFORMIX-SQL displays the previous menu or screen.

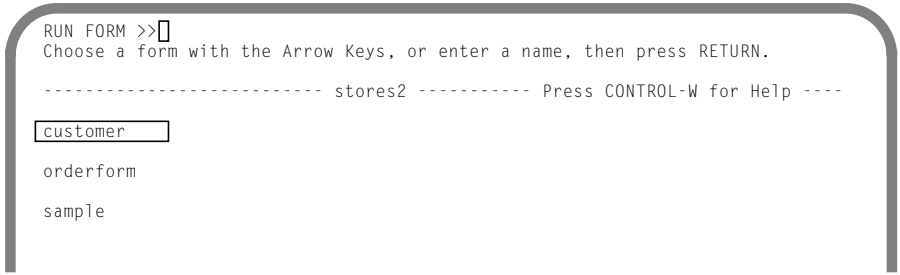
Asking for Help

The CONTROL-W key displays the help message appropriate for each part of INFORMIX-SQL. When you are finished reading the message displayed on the HELP screen, press RETURN. INFORMIX-SQL redisplays the screen you were working with before you called for help.

Text-Entry Screens

The text-entry screen is the second kind of screen. It requires that you enter text instead of choosing a menu option. The top line of the screen displays the screen name, followed by double angle (>>) brackets. The second line gives directions.

The RUN FORM screen is an example of a text-entry screen. Some of the items it includes follow.



Entering Text

Whatever you type appears after the double angle brackets at the top of the screen. Press the RETURN key when you are finished typing. Some screens, like the RUN FORM screen, give you the option of selecting an item from a list on the lower part of the screen instead of typing your selection. Use the Arrow keys to position the highlight over the item you want, and then press RETURN. INFORMIX-SQL displays the next screen.

Exiting a Text-Entry Screen

Text-entry screens do not have an Exit option. Press CONTROL-C and INFORMIX-SQL redisplays the previous menu or screen.

Asking for Help

The CONTROL-W key works with text-entry screens exactly as it does with menu screens. When you are finished reading the Help message, press RETURN. INFORMIX-SQL redisplays the screen you were working with before you called for help.

Maps of the Menu Structure

The INFORMIX-SQL Main menu has seven options: Form, Report, Query-language, User-menu, Database, Table, and Exit. Each option on the Main menu calls a submenu, which displays options that allow you to work with a part of INFORMIX-SQL. The INFORMIX-SQL menu structure is displayed in Figure 1-1 on page 1-8 and Figure 1-2 on page 1-9.

Figure 1-1 on page 1-8 is a map of the INFORMIX-SQL menu hierarchy. This figure illustrates the options on each of the submenus available from the Main menu.

Figure 1-1
INFORMIX-SQL Menu Hierarchy

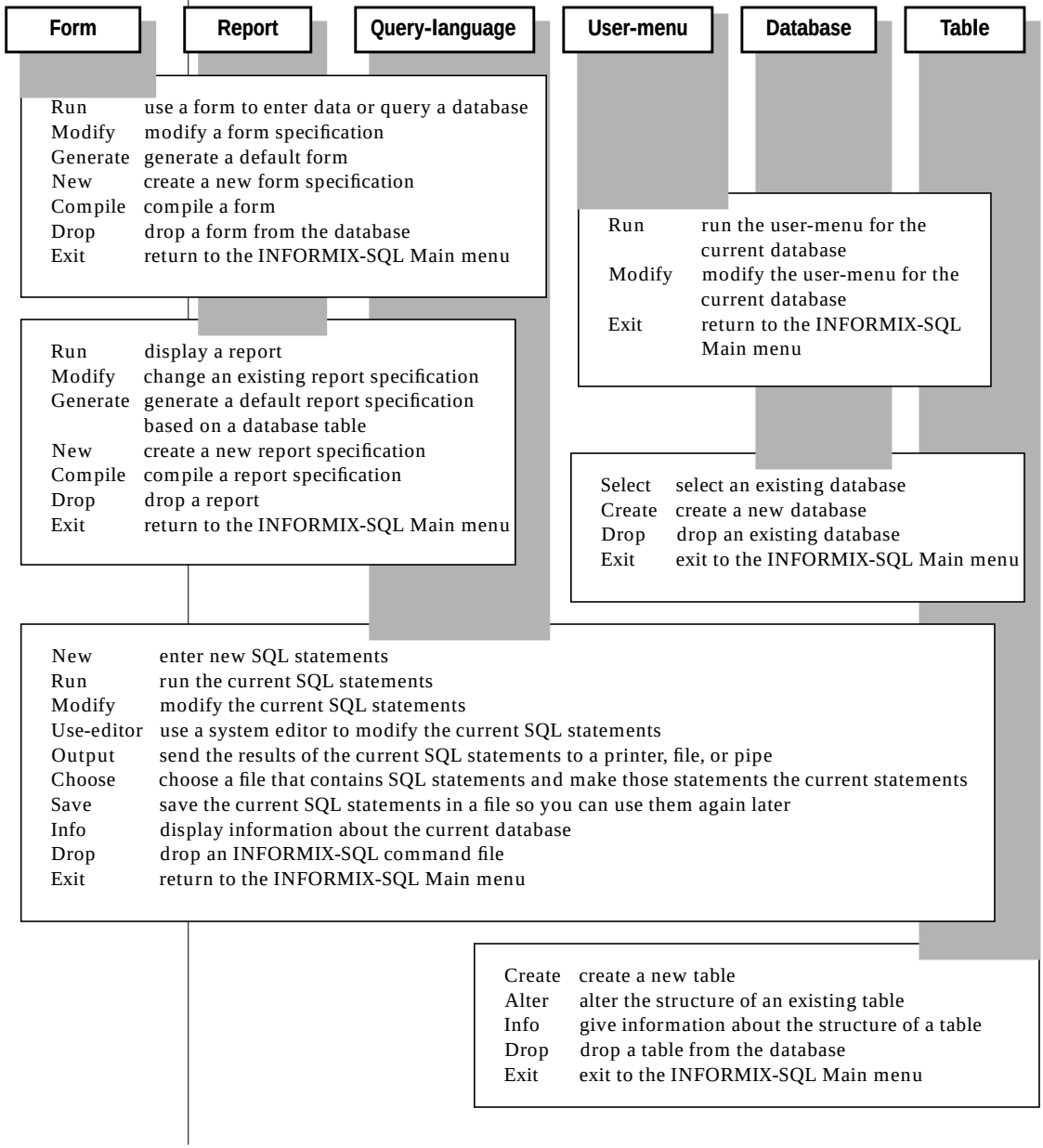


Figure 1-2 is a functional guide to the INFORMIX-SQL menu system. Menu options are grouped according to activity or task.

Figure 1-2
A Functional Guide to the Menu Hierarchy

	Menu Options					
Function	**Form**	**Report**	**Query-Language**	**User-menu**	**Database**	**Table**
Use it	Run	Run	Run	Run	Select	
Modify it	Modify	Modify	Modify	Modify		Alter
Create it						
Default	Generate	Generate	New		Create	Create
Custom	New	New	Use-editor			
Compile it	Compile	Compile				
Special Tasks			Info Choose Output Save			Info
Drop it	Drop	Drop	Drop		Drop	Drop
Exit	Exit	Exit	Exit	Exit	Exit	Exit

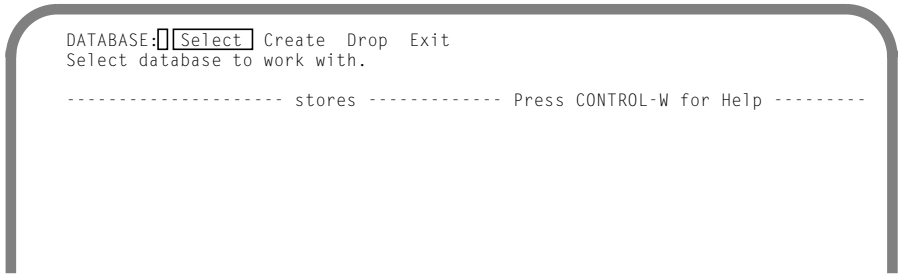
The INFORMIX-SQL Main Menu Options

The following sections discuss the options on the INFORMIX-SQL Main menu. The options are presented in alphabetical order.

DATABASE

Use the Database option to create a new database, make an existing database current, or drop an existing database (see Figure 1-3).

Figure 1-3
DATABASE Menu



Menu Options

The DATABASE menu displays four options:

Select makes a database the current database.

Create creates a new database and makes that database the current database.

Drop removes a database from the system.

Exit exits the DATABASE menu and returns to the INFORMIX-SQL Main menu.

Usage

- When you create a database with the Create option, that database becomes the current database.
- When you use the Select option, you can type the name of an existing database rather than highlight one of the database names listed on your screen. If you do so, you must enter the name of a database located in the current directory or a directory specified in your **DBPATH** environment variable. If you enter the name of a nonexistent database or a database that INFORMIX-SQL cannot locate, INFORMIX-SQL displays the following messages:

```
329:Database not found or no system permission.
2:  No such file or directory
```

- Be careful when you drop a database; all data in the database is permanently discarded.
- The *Informix Guide to SQL: Syntax* explains the workings of all SQL database statements that Informix products support.
- When using the Query-language option, you are not allowed to drop the current database. You must explicitly close it first with the CLOSE DATABASE statement. For details about the CLOSE DATABASE statement, see the *Informix Guide to SQL: Syntax*.
- For more information on the DATABASE menu, see the *INFORMIX-SQL User Guide*.

EXIT

Use the Exit option to leave the INFORMIX-SQL Main menu and return to the operating system.

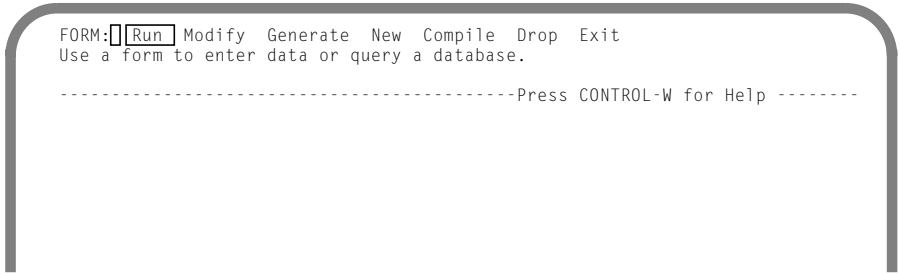
To exit from the Main menu

1. From the INFORMIX-SQL Main menu, type e to select the Exit option.
2. You leave the INFORMIX-SQL Main menu and return to the operating system.

FORM

Use the Form option to run a screen form, create or modify a screen form, compile a screen form, or drop an existing screen form (see Figure 1-4).

Figure 1-4
FORM Menu



Menu Options

The FORM menu displays the following seven options:

Run	runs a previously compiled screen form.
Modify	modifies a screen form specification.
Generate	creates a default screen form.
New	creates a custom screen form specification.
Compile	compiles a screen form specification.
Drop	drops a screen form.
Exit	exits the FORM menu and returns to the INFORMIX-SQL Main menu.

Usage

- After you edit a form specification file (with the New or Modify options on the FORM menu), you must compile it. (You cannot use the form in INFORMIX-SQL until it has been compiled.) Menus allowing you to compile an edited form are displayed when you select the New or Modify options. You can also use the Compile option on the FORM menu to compile a form specification.

  INFORMIX-SQL notifies you if errors are in the form specification. Follow the directions on the screen to correct and recompile the form. You can save or discard the form after compilation. You can also save an uncompiled form to work on at a later time or discard it completely.

- The INFORMIX-SQL program that compiles a form specification is called FORMBUILD. See Chapter 2, "The FORMBUILD Transaction Form Generator," for information about FORMBUILD.

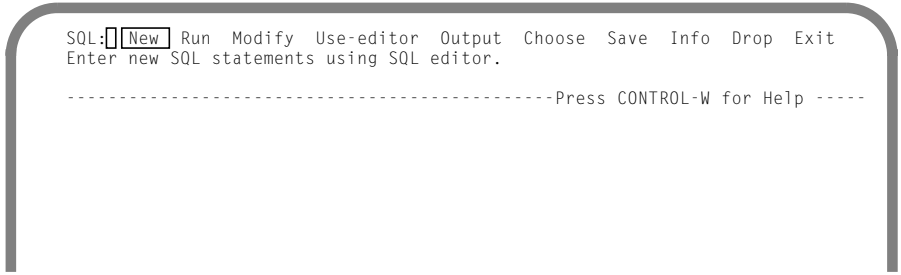
- The INFORMIX-SQL program that runs a screen form is called PERFORM. See Chapter 3, "The PERFORM Screen Transaction Processor," for information about PERFORM.

- For more information on the FORM menu, see the *INFORMIX-SQL User Guide*.

QUERY LANGUAGE

Select the Query-language option to use the SQL query language, as Figure 1-5 shows.

Figure 1-5
SQL menu



Menu Options

The SQL menu displays the following ten options:

New	allows you to enter new SQL statements using the SQL editor.
Run	executes the current SQL statement or statements.
Modify	allows you to use the SQL editor to modify the current SQL statement or statements.
Use-editor	allows you to enter or edit SQL statements with a system editor.
Output	routes the output from executing the current SQL statements to a system file, a printer, or a system pipe.
Choose	allows you to select an existing command file that contains SQL statements and make them your current statements. You can run or edit the current statements.

Save	saves the current SQL statements in a command file. You can use this command file later by selecting the Choose option on the SQL menu.
Info	allows you to retrieve information about the columns, indexes, privileges, and status of a table.
Drop	drops a command file from the database.
Exit	exits the SQL menu and returns to the INFORMIX-SQL Main menu.

Usage

- If there is no current database, INFORMIX-SQL displays the CHOOSE DATABASE screen after you select the Query-language option on the INFORMIX-SQL Main menu.
- In addition to the tables listed, you can request information about external tables if you are using Informix Dynamic Server. To specify an external table, you must enter the expanded table name at the prompt. For example, the following entry requests information from the **richard.customer** table in the **stores7** database that accesses the INFORMIX-OnLine system called **central**:

  ```
  INFO FOR TABLE >> stores7@central:richard.customer
  ```

  You can also use synonyms in place of the extended table name.

  If you select the Status option of the INFO menu, INFORMIX-SQL displays information on the dbspace that contains the table. The Status option does not display audit trail information because the logging facility replaces audit trails.
- The *INFORMIX-SQL User Guide* describes how to use the SQL menu and how to create and run SQL statements.
- The *Informix Guide to SQL: Syntax* explains the SQL database statements that Informix products support.

The following sections give some special notes about using SQL statements with the VARCHAR, TEXT, and BYTE data types.

Querying VARCHAR, TEXT, and BYTE Data

The INFORMIX-SQL Interactive Editor displays the results of a query in a format that depends on the data type of the selected column. If you execute a query on a VARCHAR column, INFORMIX-SQL displays the entire VARCHAR value, just as it displays CHAR values. If you select a TEXT column, INFORMIX-SQL displays the contents of the TEXT column and you can scroll through the contents one screen at a time by using the Next option. If you select a BYTE column, INFORMIX-SQL displays the words <BYTE value>.

Using the CREATE TABLE and ALTER TABLE Statements with Blobs

When you use the CREATE TABLE and ALTER TABLE statements, you can place quotes around blobspace names as shown in the following example:

```
CREATE TABLE mytab (column1 TEXT IN "blob1")
```

In this case, the quotes are optional. However, if the name of your blobspace is **table**, INFORMIX-SQL requires the quotes to distinguish the blobspace name with the keyword TABLE. This is demonstrated in the following ALTER TABLE statement:

```
ALTER TABLE mytab ADD (column1 TEXT IN "table")
```

In this case, the quotes are required to avoid any ambiguity with the keyword TABLE.

Using the LOAD and UNLOAD Statements with VARCHARs and Blobs

You can use the LOAD and UNLOAD statements to transfer data between a table and an operating-system file of ASCII data. This file contains only printable ASCII and newline characters.

You can use these statements on tables and files that contain the VARCHAR, TEXT, and BYTE data types. You should read these sections if you are loading or unloading files that contain VARCHAR or blob data.

For more information on using the LOAD and UNLOAD statements, see the *Informix Guide to SQL: Syntax*.

UNLOAD Statement

If you are unloading files that contain VARCHAR, TEXT, or BYTE data types, note the following information:

- BYTE items are written in hexadecimal dump format with no spaces or newline characters. Thus the logical length of an unloaded file that contains BYTE items can be very long, and it might be impossible to print or edit such a file.
- Trailing blanks are retained in VARCHAR columns.
- Do not use the following characters as delimiting characters in an unload file:
  - 0-9
  - a-f
  - A-F
  - space
  - tab
  - \

LOAD Statement

If you are loading files that contain VARCHAR, TEXT, and BYTE data types, note the following information:

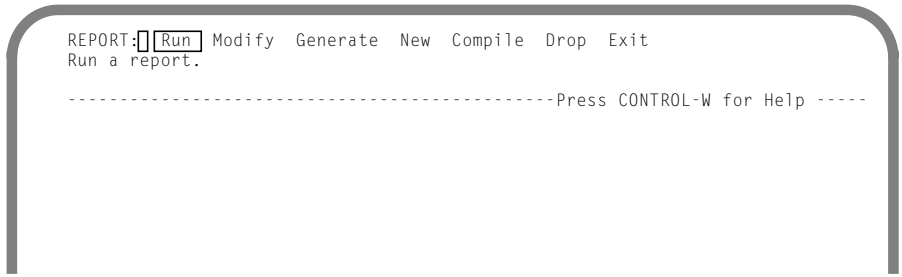
- You can give the LOAD statement data in which the character (including VARCHAR) fields are longer than the column size; the excess characters are disregarded.
- You can have leading and trailing blanks in noncharacter fields, except BYTE fields.
- In all character fields (including VARCHAR and TEXT), embedded delimiter and backslash characters are escaped with the backslash.
- In VARCHAR columns, you must escape newline characters.
- Data being loaded into a BYTE column must be in ASCII-hexadecimal form. BYTE columns cannot contain preceding blanks.

- Do not use the following characters as delimiting characters in a load file:
  - 0-9
  - a-f
  - A-F
  - space
  - tab
  - \

REPORT

Use the Report option to run a report, create or modify a report, compile a report, or drop an existing report from the database, as Figure 1-6 shows.

Figure 1-6
REPORT menu



Menu Options

The REPORT menu displays the following seven options:

Run	runs a report.
Modify	modifies a report specification.
Generate	creates a default report specification.
New	creates a custom report specification.
Compile	compiles a report specification.
Drop	drops a report specification from the database.
Exit	exits the REPORT menu and returns to the INFORMIX-SQL Main menu.

Usage

- After you edit a report specification file (with the New or Modify options on the REPORT menu), you must compile it. (You cannot use the report in INFORMIX-SQL until it has been compiled.) Menus allowing you to compile an edited report are displayed when you select the New or Modify options. You can also use the Compile option on the REPORT menu to compile a report specification.

  INFORMIX-SQL notifies you if there are errors in the report specification. Follow the directions on the screen to correct and recompile the report. You can save or discard the report after compilation. You can also save an uncompiled report to work on later or discard the report completely.

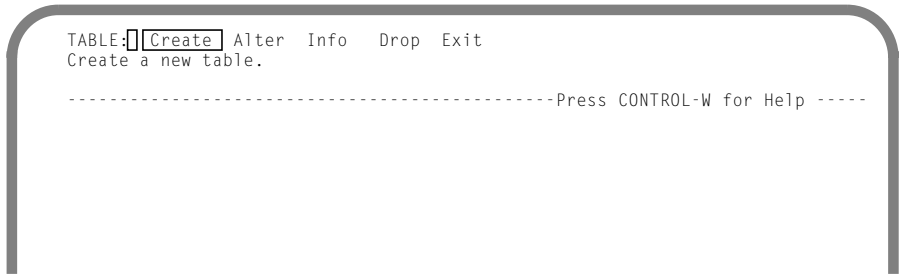
- The INFORMIX-SQL program that compiles a report specification is called ACEPREP. The INFORMIX-SQL program that runs a report specification is called ACEGO. See Chapter 4, "The ACE Report Writer," for complete information about these programs.

- The *INFORMIX-SQL User Guide* describes how to create and use reports.

TABLE

Use the Table option to create or modify a table, retrieve information about a table, or drop a table from the database, as Figure 1-7 shows.

Figure 1-7
TABLE menu



Menu Options

The TABLE menu displays the following five options:

Create allows you to use the interactive schema editor to create a new table.

Alter allows you to modify a table using the interactive schema editor.

Info retrieves information about the structure of a table.

Drop deletes a table from the database.

Exit exits the TABLE menu and returns to the INFORMIX-SQL Main menu.

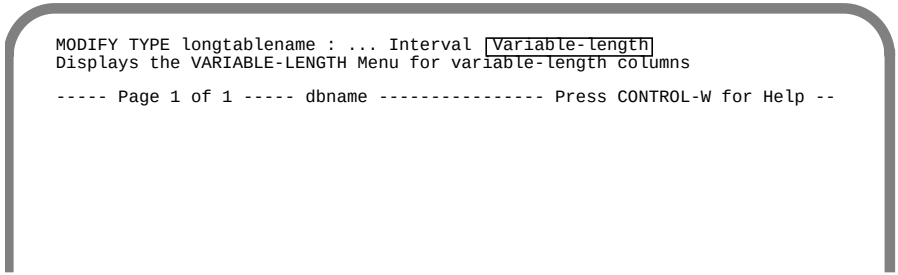
Usage

- If there is no current database, the CHOOSE DATABASE screen appears after you select the Table option.
- Be careful when you drop a table. You lose all the data in the table.
- The *INFORMIX-SQL User Guide* describes the use of options on the TABLE menu.
- The *Informix Guide to SQL: Syntax* explains the workings of all SQL database statements supported by Informix products.

Using the Table Option with Informix Dynamic Server

If you use the Table option to create a table, you get only the default size for initial and next extents. If you want other sizes, you must use the INFORMIX-SQL Interactive Editor to execute the CREATE TABLE statement that contains the explicit extent sizes.

When you are connecting to a database server, the ADD or MODIFY TYPE menu includes an additional choice of Variable-length as shown in the following screen.



If you select Variable-length, the menu shows types unique to Informix Dynamic Server, as shown here.

```
VARIABLE-LENGTH: Varchar  Text  Byte
Variable-length data containing a maximum of 255 characters.

----- Page 1 of 1 ----- dbname ---------------- Press CONTROL-W for Help --
```

You can select any of these types to set up a variable-length column in your table.

If you select the VARCHAR data type, you are prompted for the column length. A VARCHAR column has two lengths: a maximum size and a minimum space. You can specify these two numbers at the subsequent prompts, as shown in the following screens.

```
ADD MAXIMUM LENGTH >>
Enter maximum length of data from 1 to 255.  RETURN adds it.

----- Page 1 of 1 ----- dbname ---------------- Press CONTROL-W for Help --
```

```
ADD MINIMUM SPACE >>
Enter amount of space to reserve for each item from 0 to max length.

----- Page 1 of 1 ----- dbname ------------------ Press CONTROL-W for Help -
```

If you select either TEXT or BYTE, you must indicate where the data is stored. The BLOBSPACE menu is shown here.

```
ADD BLOBSPACE tab1: [Table] BLOBSpace-name
Column data is stored in the same table-space as other columns.

----- Page 1 of 1 ----- dbname ---------------- Press CONTROL-W for Help -
```

If you choose Table, the column data is stored in the same dbspace as the other columns. If you choose BLOBSpace-name, you see the following prompt.

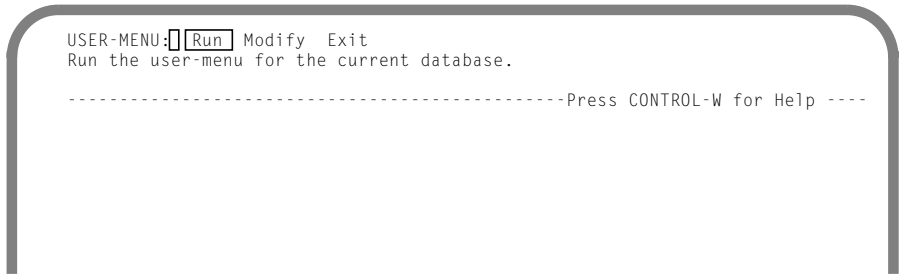
```
ADD BLOBSPACE NAME >>
Enter name of BLOBSpace

----- Page 1 of 1 ----- dbname --------- Press CONTROL-W for Help -
-
```

You can enter the name of any existing blobspace at the prompt.

USER MENU

Use the User-menu option to run a user-created menu, create a user-menu, or modify an existing user-menu as Figure 1-8 shows.

Figure 1-8
USER-MENU menu



Menu Options

The USER-MENU menu displays the following three options:

Run runs the user-menu for the current database.

Modify allows you to create or modify a user-menu.

Exit exits the USER-MENU and returns to the INFORMIX-SQL Main menu.

Usage

- After you select the User-menu option, the CHOOSE DATABASE screen appears if there is no current database.
- Use the Modify option to both create and modify a user-menu.
- If there is no User-menu for the current database, INFORMIX-SQL displays a message notifying the user when the Run or Modify option is selected.
- See Chapter 5, "User-Menu," for more information about creating, modifying, and using a menu.

Chapter

2

The FORMBUILD Transaction Form Generator

In This Chapter

Before you can use PERFORM with a customized screen form (see Chapter 3, "The PERFORM Screen Transaction Processor"), you must first use FORMBUILD to compile a form specification file. The form specification file contains the screen format and the instructions to PERFORM about how to display the data.

PERFORM Error Messages

The text of all error messages and suggestions for corrections is included in *Informix Error Messages* in Answers OnLine.

Sample Forms

Examples that are in the *INFORMIX-SQL User Guide* and the *INFORMIX-SQL Reference Manual* are based on the following five sample form specifications. These form specifications illustrate a wide variety of commands available with PERFORM.

customer.per	A simple form used to enter and retrieve customer information.
orders.per	A simple form used to enter and retrieve order information.
orderform.per	A more complex form used to enter and retrieve information about customer orders.
sample.per	The most sophisticated form included with the stores7 database. It is an expanded version of the ORDERFORM form.
p_ex1.per	Illustrates calling a C function from within a form.

Appendix A, "The Demonstration Database and Examples," contains the full text of these sample form specifications. In addition, a copy of the **sample** specification is included in the section entitled "The SAMPLE Form Specification File" on page 2-92.

Creating and Compiling a Custom Form

You can create a form specification file in one of two ways: you can use the Form option on the INFORMIX-SQL Main menu, or you can work directly with the appropriate programs from the operating system command line. Either alternative requires that you have already created the database and all the tables to which the form will refer. The following two sections describe these alternative procedures. They do not, however, describe the rules on how to construct or modify the form specification file. These rules are defined in the remaining sections of this chapter.

Using the Menu System to Create a Form

To create a customized screen form using the INFORMIX-SQL menu system, follow these steps:

1. Select the Form option on the INFORMIX-SQL Main menu and then the Generate option on the FORM menu.
2. If there is no current database, the SELECT DATABASE screen appears. After you select a database, the GENERATE FORM screen displays. Enter the name you want to assign to the form (for example, NEWFORM). INFORMIX-SQL asks you for the names of the tables whose columns you want in your form. The GENERATE FORM menu allows you to enter up to eight tables. (If you want to include more than eight tables in your form specification, use the New option on the FORM menu to create it from scratch.) When you have selected all the tables you want to include, FORMBUILD creates a default form specification file. The FORM menu then displays. You can now use the default screen form with PERFORM.

The default form specification file formats the screen as a list of all the columns in the tables included in the form. It does not provide any special instructions to PERFORM about how to display the data, nor does it include instructions to perform data manipulations.

3. Select the Modify option on the FORM menu, and INFORMIX-SQL displays the MODIFY FORM screen. Indicate the name of the default form specification (NEWFORM). If you have not specified an editor previously in this session or set the DBEDIT environment variable (as explained in Appendix B, "Setting Environment Variables"), INFORMIX-SQL asks for the name of your editor. Then INFORMIX-SQL calls your system editor with the file.

   Edit the default form specification file to produce your customized screen form and associated instructions. Exit from the editor.

4. The MODIFY FORM menu displays. Select the Compile option.

5. If your form specification file compiles correctly, a message to that effect displays, and FORMBUILD creates a form file with the filename extension **.frm** (for example, **newform.frm**). Go to step 7. If your form specification file contains errors, a message to that effect displays, and FORMBUILD creates a form file with the filename extension **.err** (for example, **newform.err**). Go to step 6.

6. Select the Correct option from the COMPILE FORM menu. INFORMIX-SQL calls your system editor with the form specification file marked with the compilation errors. When you correct your errors, you need not delete the error messages. INFORMIX-SQL does that for you. Repeat step 4.

7. When the compilation is successful, select the Save-and-exit option on the MODIFY FORM menu.

As an alternative to using the Generate option and creating a default form specification, you can select the New option. INFORMIX-SQL calls your system editor.

The Generate option is usually a more efficient way to create a custom form because, if you use the New option, you must enter all form specification instructions into the file.

Using the Operating System to Create a Form

To create a customized screen form directly from the operating system command line, follow these steps:

1. Create a default form specification file by entering the following command at the operating-system prompt:

   ```
   sformbld -d
   ```

   FORMBUILD asks for the name of your form specification file, the name of your database, and the names of the tables whose columns you want in your form. FORMBUILD allows you to enter up to 14 tables. When you enter a blank line for the table name, FORMBUILD assumes you have selected all the tables and creates a default form specification file. FORMBUILD appends the extension **.per** to the name of the file.

   Alternatively, you can create a form specification file using a system editor. You do not need to add the **.per** extension to the name of the form file, but you can if you want. If you use this method, proceed to step 3.

2. Use a system editor to modify the default form specification file to meet your specifications.

3. Enter the command

   ```
   sformbld newform
   ```

   where **newform** is the name of your form specification file (without the **.per** extension). If the compilation is successful, FORMBUILD creates a compiled form file called **newform.frm**, and you are finished creating your customized screen form. If your compilation is not successful, FORMBUILD creates a file named **newform.err**, and you must proceed to step 4.

4. Edit the file **newform.err** and correct the compilation errors. You must erase the error messages. Overwrite the file **newform.per** with this corrected version and repeat step 3.

To run the compiled form specification directly from the command line, enter the following command:

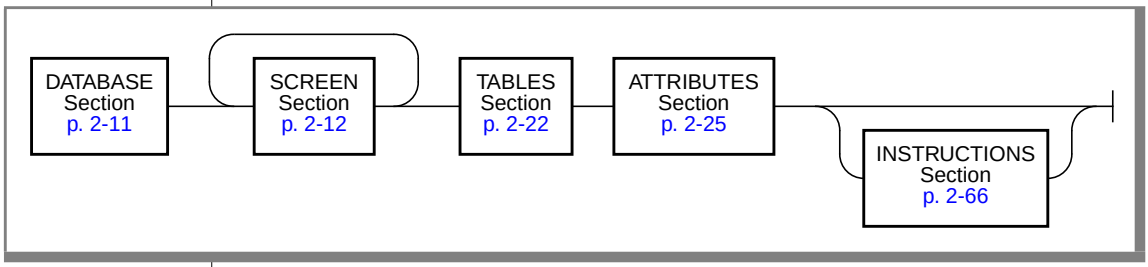
```
sperform newform
```

You can also create a customized screen form from the operating system command line using a shortened version of the INFORMIX-SQL Main menu options. Appendix G, "Accessing Programs from the Operating System," discusses this method in detail.

Structure of a Form Specification File

Form specification files consist of four required sections (DATABASE, SCREEN, TABLES, and ATTRIBUTES) and one optional section (INSTRUCTIONS) as shown in Figure 2-1.

Figure 2-1
Syntax of a Form Specification File



DATABASE section	Each form specification file must begin with a DATABASE section that identifies the database you want to use with the form.
SCREEN section	The SCREEN section appears next and shows the exact layout of the form as you want it to appear on the screen. If the form has several screens, this section includes the layout for each screen, one after another. You can use graphics characters to enhance the appearance of the screen.
TABLES section	Each form specification file must contain a TABLES section following the SCREEN section. The TABLES section identifies the tables whose columns appear in the form.
ATTRIBUTES section	The ATTRIBUTES section describes each field on the form including, for example, appearance, acceptable input values, displayed comments, and default values.
INSTRUCTIONS section	The INSTRUCTIONS section is optional and specifies master-detail relationships, composite joins, alternative field delimiters, and control blocks.

Using the END keyword to mark the end of sections in the form specification file is optional. Some users find it helpful to indicate the close of a section with END. The forms included with the demonstration database use the END keyword.

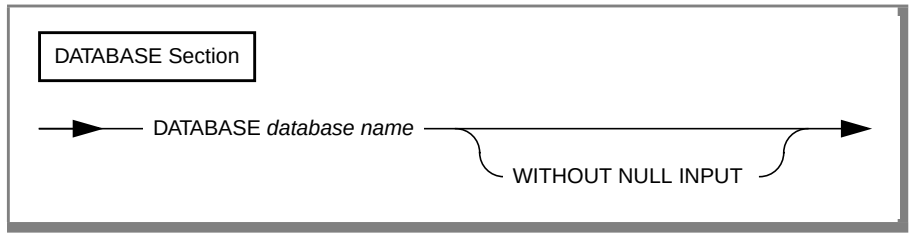
Figure 2-2 illustrates the overall structure of a form specification file.

Figure 2-2
A Partial Form Specification File

```
database stores7

screen
{
--------------------------------------------------------------------------------
CUSTOMER INFORMATION:
Customer Number: [c1         ]             Telephone: [c10              ]
            .
            .
            .

SHIPPING INFORMATION:
     Customer P.O.: [o20       ]

          Ship Date: [o21       ]          Date Paid: [o22       ]
}

end

tables
customer orders

attributes
c1 = *customer.customer_num
   = orders.customer_num;
c10 = phone, picture = "###-###-####x#####";
            .
            .
            .
o20 = po_num;
o21 = ship_date;
o22 = paid_date;

instructions
customer master of orders;
orders master of items;
end
```

DATABASE Section

The DATABASE section of a form specification file identifies the database with which the form is designed to work.



DATABASE	is a required keyword.
database-name	is the name of the database.
WITHOUT NULL INPUT	are optional keywords that enable you to disallow NULL values.

Use the WITHOUT NULL INPUT option only if you have elected to create and work with a database that does not have NULL values. For fields that have no other defaults, this option causes INFORMIX-SQL to display *zeros* as default values for number and INTERVAL fields, and *blanks* for character fields.

The default DATE value is `12/31/1899`; the default DATETIME value is `1899-12-31 23:59:59.99999`.

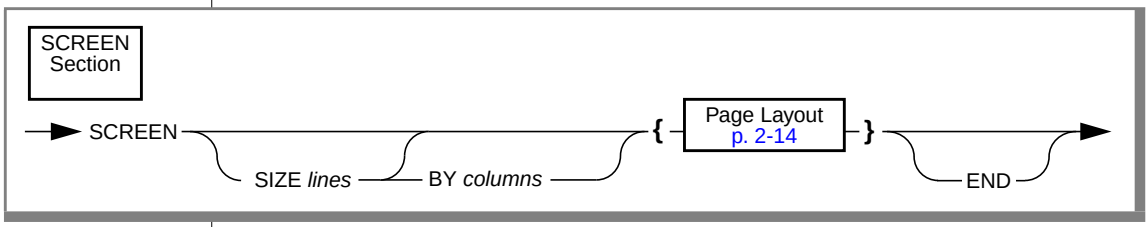
The following DATABASE section is from the **sample** form specification file at the end of this chapter:

```
database
    stores7
```

With Informix Dynamic Server, you can specify the full name of a remote database in the DATABASE section. List the simple names of the tables in the TABLES section. For more information, see "TABLES Section" on page 2-21.

SCREEN Section

The SCREEN section of the form specification file describes how the form appears on the screen when you use it with PERFORM. A form specification can include multiple SCREEN sections that correspond to multiple page layouts.



SCREEN	is a required keyword.
SIZE	is an optional keyword that tells FORMBUILD to create a screen that is a specific number of *lines* long and *cols* wide.
lines	is an integer that specifies the screen length in lines. The default is 24 lines.
BY	is an optional keyword.
cols	is an integer that specifies the screen width in columns. The default is 80 columns.
END	is an optional keyword to end the SCREEN section.

Usage

- Each page layout is preceded by the SCREEN keyword and is enclosed in braces ({ }). A page layout consists of an array of *display fields* and textual information, such as titles, field labels, and graphics characters. Display fields are indicated by brackets ([ ]) that define the field length and by field tags that identify the field.
- The default SCREEN section is `SCREEN SIZE 24 by 80`. FORMBUILD prepares a screen of up to 20 lines (4 lines are reserved for system use) and up to 80 characters in a line.
- Use the SIZE keyword to indicate an alternative screen size. If you do not indicate a larger screen size, and if you include more than 20 screen lines between a pair of braces, FORMBUILD splits the page, with line 21 at the top of the second page.

- If you specify a screen size, the size must appear on the first screen. The size applies to all the screens.
- You can use command-line syntax to override either or both of the lines or *dimensions* of the SCREEN section by specifying:

  ```
  sformbld -l  lines -c  cols filename
  ```

  where *lines* and *cols* are defined as in the above syntax diagram, and *filename* is the name of the form specification file. FORMBUILD uses the **INFORMIXTERM** environment variable to determine whether to use **termcap** or **terminfo** at compile time to set screen characteristics. If **INFORMIXTERM** is unset, FORMBUILD uses **termcap**.

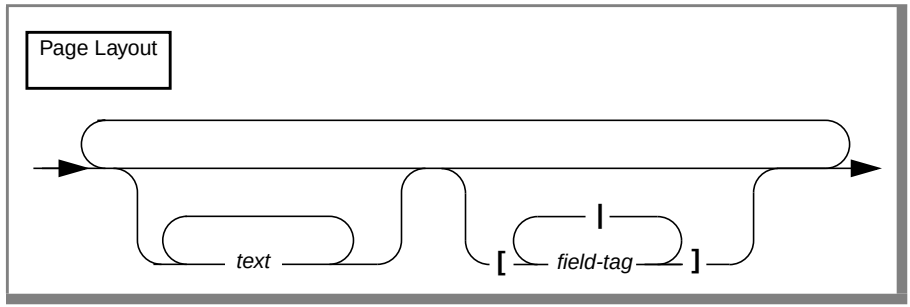
The following example illustrates the use of the SCREEN keyword with multiple page layouts:

```
SCREEN SIZE 18 BY 75
{

     .
     .
     .
display fieldspage layout
     .
     .
     .

}

SCREEN
{
     .
     .
     .
display fieldspage layout
     .
     .
     .

}
```

The **sample** form included at the end of this chapter demonstrates the SCREEN sections of a multiple-page form.

Page Layout

You indicate where data is to be displayed on the screen by using brackets ([ ]) to delimit a field. Each field has an associated field tag that identifies the field in the ATTRIBUTES and INSTRUCTIONS sections.



text	is the material you want displayed on the screen.
[ ]	are delimiters for a field. The width of the field is the number of characters that can placed between the brackets. In this context, the brackets do not signify an optional input.
|	denotes the close of one field and the beginning of the next field.
field-tag	is the field tag used to identify the display field.

Usage

- Each field must have a field tag. The field tag is from 1 to 50 characters long. The first character must be a letter; the rest of the tag can include letters, numbers, and underscores (_). The field tag must be short enough to fit within the brackets. You can use the same field tag at more than one position in the SCREEN section of the form specification if you want the same column information to appear in more than one place.
- Field tags are not the same as database column names; they are the associations used in the SCREEN section, the ATTRIBUTES section, and the INSTRUCTIONS section to tell PERFORM where to display and store information. The ATTRIBUTES section associates each field tag with a column in your database or identifies it as a display-only field.
- FORMBUILD ignores the case of a field tag; **a1** and **A1** are the same.

- One-character columns are given the display tags **a** through **z**. This means that a screen form can use no more than 26 one-character columns.
- When you create a default form specification file, the widths of all fields are determined by the data type of the corresponding columns in the database tables.
- The width of a field in the SCREEN section of the form specification is normally set equal to the width of the database column to which it corresponds. You can reduce or expand the screen width, but you should be careful because this can truncate the screen presentation or database storage of the data.
- Fields corresponding to numeric columns should be large enough to contain the largest number that you might display. If the field is too small to display the number you assign to it, PERFORM fills the field with asterisks.
- Fields for BYTE data are never displayed; the phrase <BYTE value> is shown in the display field to indicate that the user cannot see the BYTE data. The following excerpt from a form specification shows a TEXT field **resume** and a BYTE field **photo**. You do not need to have more than one line on a form for a TEXT field. The BYTE field is short because only the words <BYTE value> are displayed.

```
resume  [f003                                          ]
photo   [f004        ]
attributes
f003 = employee.resume
f004 = employee.photo
```

- Fields for CHAR type data can be shorter than the defined length of the column. PERFORM fills the field from the left and truncates the right end of any longer CHAR string assigned to the field. Through subscripting, you can assign portions of a CHAR column to one or more fields. (See the "ATTRIBUTES Section" on page 2-24.)

- If you edit and modify the default form specification file or create one from scratch, you can verify that the character column field widths match the data type of the corresponding columns by using the verify (**-v**) option of FORMBUILD. Enter the following command in response to the system prompt:

  ```
  sformbld -v newform
  ```

  FORMBUILD reports any discrepancies in the file *newform*.**err**, where *newform* is the name of the form specification file that has been verified.

- The | bar symbol can be used to denote the close of one field and the beginning of the next field. In the following example, *field-tag1* identifies the first display field; *field-tag2* identifies the second display field:

  ```
  text [field-tag1 | field-tag2 ]
  ```

  When you use the bar symbol to denote the close of one field and the beginning of the next field, you must include a DELIMITERS statement in the INSTRUCTIONS section of the form specification. Use the same symbol as both the left and right delimiters in the statement.

The screen layout from the **sample** form specification file follows:

```
screen
{
================================================================================
================================================================================

Customer Number: [c1         ]
        Company: [c4                  ]
     First Name: [c2        ]             Last Name: [c3             ]
        Address: [c5                  ]
                 [c6                  ]
           City:  [c7             ]  State: [c8]   Zip: [c9   ]
     Telephone:  [c10              ]

================================================================================
================================================================================
}
screen
{
================================================================================
CUSTOMER NUMBER: [c1         ]           COMPANY: [c4                  ]

ORDER INFORMATION:
     Order Number: [o11         ]       Order Date: [o12          ]

     Stock Number: [i13        ]      Manufacturer: [i16]
      Description: [s14            ]                [m17            ]
             Unit: [s16            ]
                                           Quantity: [i18        ]
                                          Unitprice: [s15               ]
                                        Total Price: [i19               ]
SHIPPING INFORMATION:
 Customer P.O.: [o20       ]           Ship Charge: [d1        ]

       Backlog: [a]             Total Order Amount: [d2        ]
     Ship Date: [o21       ]
     Date Paid: [o22       ]
  Instructions: [o23                                    ]
}
end
```

Graphics Characters in Forms

You can use graphics characters to place boxes and other rectangular shapes in a screen form.

You can enter command strings to invoke a simple line drawing or graphics in which standard characters produce special effects on a compiled, displayed form. The letter p, for example, becomes the upper left corner of a box while a series of hyphens becomes a solid horizontal rule, as shown in Figure 2-3.

Form Specification

```
database STORES7
screen
{

\gp------------------------------------------------------------q
\g|-----------------------\gCUSTOMER\g-----------------------|
\g|\g
\g|\g      Cust No  [f000] Company   [f001               ]  \g|
\g|\g                                                        \g|
\g|\g        Name:  [f002        ]    [f003           ]    \g|
\g|\g                                                        \g|
\g|\g          Telephone Number:   [f004          ]         \g|
\g|\g
\g
\g
\g
\g
}
e
```

CUSTOMER

Cust No. [112] Company [Runners & Others]

Name [Margaret] [Lawson]

Telephone Number: [415-887-7235]

Displayed Form

Figure 2-3
Form Specification and Displayed Form (Method 1)

The following procedure outlines the steps for specifying graphics characters:

1. Create a form in the usual manner.
2. Use the following characters in the SCREEN section to indicate the borders of one or more boxes on the form.

Character	Produces
p	upper left corner of box
q	upper right corner of box
b	lower left corner of box
d	lower right corner of box
_	a horizontal-line character
|	a vertical-line character

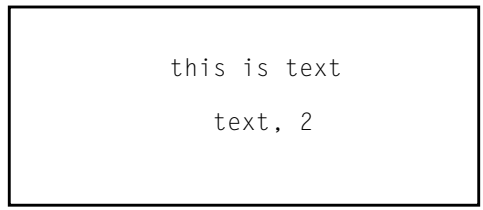
The meanings for these six characters are derived from the **gb** specification in the **termcap** file, or the **acsc** specification in the **terminfo** file. INFORMIX-SQL substitutes the graphics characters specified in the **termcap** or **terminfo** file for these characters when you display the compiled form.

3. After the form has the desired configuration, use the **\g** string to indicate when to begin graphics mode and when to end graphics mode.

   Insert the **\g** string before the first **p**, **q**, **d**, **b**, dash, or pipe that represents a graphics character. To leave graphics mode, insert the string **\g** after the **p**, **q**, **d**, **b**, dash, or pipe. Figure 2-4 shows the commands that draw a box around text.

```
\gp----------------------q
\g|\g                     \g|
\g|\g     this is text    \g|
\g|\g                     \g|
\g|\g        text, 2      \g|
\g|\g                     \g|
\gb----------------------d
```

Specification



Displayed Result

Figure 2-4
Drawing a Box Around Text

Do not insert the **\g** strings in original white space on the form. The backslash should displace the first graphics character in the row and push the remaining row to the right. Although this distorts the way the form specification looks on screen, the actual output will not be distorted.

In your form specification, you can include not only the characters used to create a box or rectangle, but also other graphics characters. However, the meaning of a character other than **p**, **q**, **d**, **b**, dash, and pipe depends on your terminal.

Required Terminal Entries

If you plan to use graphics characters, your terminal entry in the **termcap** or **terminfo** files must include the following variables:

termcap:

gs the escape sequence for entering graphics mode.

ge the escape sequence for leaving graphics mode.

gb the concatenated, ordered list of ASCII equivalents for the six graphics characters used to draw the border.

terminfo:

smacs the escape sequence for entering graphics mode.

rmacs the escape sequence for leaving graphics mode.

acsc the concatenated, ordered list of ASCII equivalents for the six graphics characters used to draw the border.

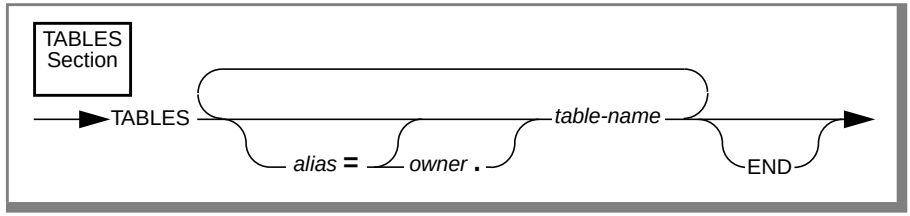
For information about making changes to your **termcap** and **terminfo** files, see Appendix B, "Setting Environment Variables," or check the manual that comes with your terminal.

TABLES Section

The third section of the form specification file lists all the tables from which columns appear in the screen form. You need not display in the screen form every column of every table listed, but any table contributing data to the form must be included.

The TABLES section consists of the TABLES keyword, followed by a table name or a list of table names separated by spaces.

In an ANSI-compliant database, a form must qualify any table name with the *owner.* prefix if users other than *owner* run the form. If you specify an owner name, you must specify a simple alias for *owner.table-name* in the TABLES section to reference the table in other sections of the form specification file.



TABLES	is a keyword to begin the TABLES section.
alias	is the table name, synonym, or alias for the name of the table in the form specification file.
owner	is the username of the user who created the table.
table-*name*	is the identifier or synonym of the table in its database.
END	is an optional keyword to end the TABLES section.

Usage

- You cannot include a temporary table in your table list.
- You can build a form based on a view as long as the columns that contribute data to the view belong to only one table. Aggregate data is not allowed.
- The number of tables that you can use in a form is machine dependent. On most UNIX systems, the maximum number of tables open at one time is 12.

The TABLES section from the **sample** form specification file at the end of this chapter is as follows:

```
tables
    customer items stock
    orders manufact
```

The following TABLES section specifies aliases for two tables:

```
tables
    tab1 = refdept.archive
    tab2 = athdept.equip
```

Remote Databases

With Informix Dynamic Server, you can specify remote databases, external tables, and external, distributed tables in forms. You can make a remote database the current database for use with the form, or you can specify a table external to the current database in your form.

You can specify the full name of a remote database in the DATABASE section of the form. List the simple names of the tables in the TABLES section. You can also use table-name synonyms in the TABLES section as long as they have been defined for the current database. These synonyms can stand for tables in the current database or in other databases.

In the ATTRIBUTES section of the form, refer to tables in the current database with their simple table names. You can also use synonyms in the ATTRIBUTES section as follows:

```
ATTRIBUTES
    f0 = table-name.colname, .....;
    f1 = synonym.colname,  ......;
    f2 = view-name.colname, .....;
    .
    .
    .
```

External Tables and Synonyms

If you want to use a table or synonym that has a multipart name, you must first give it an alias. Examples of objects that require table aliases follow:

- External tables
- Tables within the current database that are qualified by their owner names
- Multipart synonyms

Use the following syntax to give an object a table alias:

```
TABLES
.
.
.
table-alias=database@servername:[owner.]table
table-alias=[owner.]table
```

The table-alias is a single-word identifier.

In the ATTRIBUTES and INSTRUCTIONS sections of the form, refer to external tables in the following way:

```
ATTRIBUTES
     .
     .
     .
     f3 = table-alias.colname, .....;
     .
     .
     .
```

For example, if you have the following declaration in your tables section:

```
timecard_a = otherdb:acctg.timecard
```

part of your attributes section might look like this:

```
f3 = timecard_a.sickleave
```

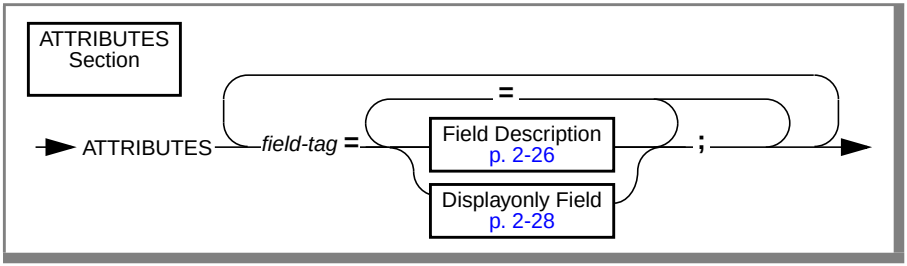
You see the alias rather than the actual table name in error messages. If you are using PERFORM, you see the alias instead of the table name on the second line of the screen.

ATTRIBUTES Section

The ATTRIBUTES section describes the behavior and appearance of each field defined in the SCREEN section. Every field in the SCREEN section must be described in the ATTRIBUTES section. You use *attributes* to describe how PERFORM should display the field, to specify a default value, to limit the values that can be entered, and to set other parameters, as described in the "ATTRIBUTES Syntax" on page 2-31.

The order in which the fields are described in the ATTRIBUTES section determines the default order for the cursor movement on the screen. The order in which columns are referenced determines the order in which PERFORM makes tables active (that is, available for data entry).

Fields in the SCREEN section do not have to be associated with a column. A field not associated with a column is called a *display-only field*. The ATTRIBUTES section contains the following two kinds of *link statements*: statements that link field tags to database columns and statements that link field tags to display-only fields.



ATTRIBUTES	is a required keyword.
field-tag	is the field tag used in the SCREEN section

Display Field Order

When you use the Query, Add, and Update options in PERFORM, the cursor advances by default from field to field in the active table according to the order in which the field tags appear in the ATTRIBUTES section of the form specification. When the cursor has advanced through all the fields in the active table on the screen, it returns to the first field. You can change the default order by using control blocks, described in the "INSTRUCTIONS Section" on page 2-66.

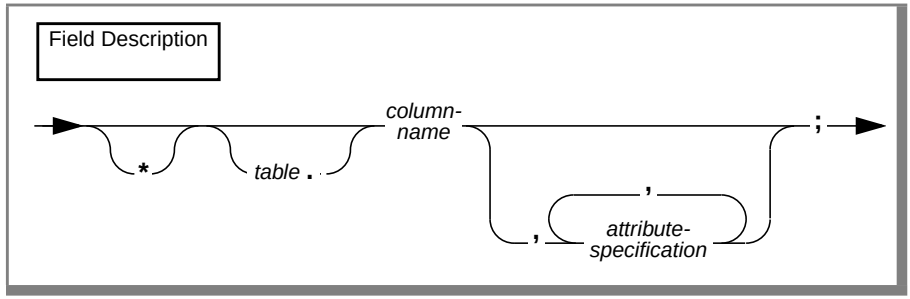
Table Order

When a form contains fields that correspond to several database tables, PERFORM puts the tables in an ordered list. When you use the Table option, PERFORM changes the active table by selecting the next table in the list. PERFORM assigns tables in the order in which columns in the tables are referenced in the ATTRIBUTES section. You reference a table whenever you associate a column from the table with a tag name or another column in a join.

Fields Linked to Database Columns

You can link two kinds of fields to database columns in the ATTRIBUTES section: fields that accept input and fields that do not. Fields that are linked to database columns and that do not allow input from the keyboard are called *lookup fields*. You specify lookup fields with the LOOKUP attribute, defined in "ATTRIBUTES Syntax" on page 2-31.

This section describes column-linked fields that allow input.



*	is optional punctuation that identifies the dominant column in a verify join.
table	is the optional name or alias of a database table. You need to specify table only if the same column name occurs in more than one table in the form. (This allows PERFORM to uniquely identify a column.)
column-name	is the name of a column.
attribute-specification	is a FORMBUILD attribute or a list of attributes separated by commas. All the attributes are described in the "ATTRIBUTES Section" on page 2-24.

Usage

- You can display portions of CHAR-type columns in a field by using subscripting. For example, the **orders** table has a **ship_instruct** column that is a CHAR-type column of length 40. You can display it on the screen as two display fields of length 20. If the field tags for the two fields are **inst1** and **inst2**, respectively, the ATTRIBUTES section entry is as follows:

```
inst1 = ship_instruct[1,20];
inst2 = ship_instruct[21,40];
```

  You can also use the WORDWRAP attribute to display long CHAR fields on multiple lines.

- If you use an alias in the TABLES section, you must use the alias to refer to the table in the ATTRIBUTES section.

- In the ATTRIBUTES and INSTRUCTIONS sections of the form, refer to external tables in the following way:

```
ATTRIBUTES
     .
     .
     .
     f3 = table-alias.colname, .....;
     .
     .
     .
```

  For example, if you have the following declaration in your tables section:

```
timecard_a = otherdb:acctg.timecard
```

  part of your ATTRIBUTES section might look like this:

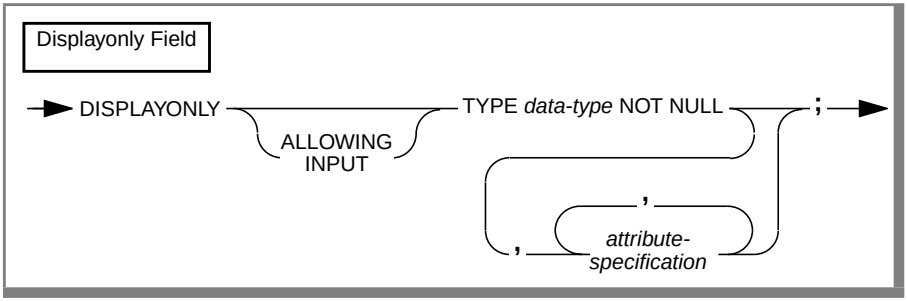
```
f3 = timecard_a.sickleave
```

  You see the alias rather than the actual table name in error messages. If you are using PERFORM, you see the alias instead of the table name on the second line of the screen.

  For more information about accessing external tables, see "TABLES Section" on page 2-21.

Display-Only Fields

Display-only fields are not associated with columns of the database and appear only on the screen. They receive their values as a result of calculations or logical decisions based on the values in other fields.



DISPLAYONLY	is a required keyword that indicates that the field does not correspond to a column of a table in the database. You describe how such a field receives its value in the INSTRUCTIONS section.
ALLOWING INPUT	are optional keywords that you use to allow input.
TYPE	is a required keyword.
data-type	is any one of the data types permitted by INFORMIX-SQL except SERIAL. (For information about data types, see *Informix Guide to SQL: Reference*.)
NOT NULL	are optional keywords that inform INFORMIX-SQL that, if the field allows input, the user must give it a value.
attribute-specification	is an attribute or a list of attributes separated by commas. All the attributes are described in the "ATTRIBUTES Section" on page 2-24.

- Do not give a length to type CHAR; the display width determines the length.
- If you specify the precision for a DECIMAL or MONEY type, be certain that the display width can hold the value.
- When the field does not allow input, you can use only the following attributes with display-only fields:

```
DEFAULT       DOWNSHIFT
FORMAT        QUERYCLEAR
REVERSE       RIGHT
UPSHIFT       ZEROFILL
```

- When you specify that one or more display-only fields allow input, PERFORM collects these fields into a database table named **displaytable**. No database table is actually created, but PERFORM behaves as though **displaytable** existed and as though its field tags were column names. You use **displaytable** in the INSTRUCTIONS section to control data entry and cursor movement in display-only fields that allow input.
- If a **displaytable** exists, it is always the last table in the sequence of active tables, regardless of how you order its fields among the other field tags.
- When the DATABASE section has the WITHOUT NULL INPUT clause, the NOT NULL keywords instruct INFORMIX-SQL to use zero (number data type) or blanks (CHAR data type) as a default for this field.
- With Informix Dynamic Server, you can use the keywords TEXT and BYTE when you define a display-only (FORMBUILD) field. There is marginal value in designating one field as type BYTE because only the words <BYTE value> are displayed.

The ATTRIBUTES section of the **sample** form specification file contains the following two DISPLAYONLY fields:

```
d1 = displayonly type money;
d2 = displayonly type money;
```

These fields are used to calculate the shipping charge and total order amount for each order. This information is not stored in any columns in the database.

Joining Columns

A screen form that contains information from several database tables normally includes a display field that *joins* two (or more) database columns that contain the same information. While it is not required that the join columns be indexed, it is advisable because cross-table queries do not run as quickly if the underlying join columns are not indexed.

The database columns you join must be of the same data type. If they are CHAR columns, they must be the same length. Do not join two SERIAL columns to each other; join a SERIAL column only to an INTEGER column.

You join columns by equating them to the same field tag in the ATTRIBUTES section:

```
field-tag = col1 = col2;
```

An example from the **sample** form follows:

```
o11 = *orders.order_num = items.order_num;
```

Field-tag **o11** joins the **order_num** column of the **orders** table with the **order_num** column of the **items** table. (The asterisk placed before the **orders.orders_num** column name indicates that this is a special kind of join—a verify join. Verify joins are explained on page 2-30.)

The placement of attributes determines when they take effect. If you want an attribute to apply regardless of which table in the join is active, place the column names on the same line and the attribute after the last column name:

```
field-tag = col1 = col2, attr;
```

If you want different attributes to apply for each of the columns in the join, place the column names on separate lines:

```
field-tag= col1, attr1;
    = col2, attr2;
```

attr1 is effective when the table that contains *col1* is active, and *attr2* is effective when the table that contains *column2* is active.

Here is an example from the **sample** form:

```
i13 = items.stock_num;
    = *stock.stock_num, noentry,
        noupdate, queryclear;
```

The attributes NOENTRY, NOUPDATE, and QUERYCLEAR (explained in "ATTRIBUTES Syntax" on page 2-31) are effective only when the **stock** table is active.

The FORMAT and REVERSE attributes, also described in the "ATTRIBUTES Syntax" section, always take effect, regardless of their placement.

Verify Joins

You can verify that the value you enter into a field that corresponds to a column in one table already exists in another column (the *dominant* column) in another table. You do this through a *verify join*. You indicate the verify join by placing an asterisk in front of the dominant column name, as follows:

```
field-tag = col1 = *col2;
```

PERFORM prevents entry of any value into *field-tag* that does not already occur in *col2*. (This applies for noncomposite conditions.)

For example, when you assign orders to customers, you want to ensure that the customer number entered for a store is a valid customer number in the **customer** table. The following statement in the ATTRIBUTES section of the **sample** form does just this:

```
c1   = *customer.customer_num
     = orders.customer_num;
```

In the previous statement, the **customer.customer_num** column is the dominant column in a verify join; when the **orders** table is active, you cannot enter a value into field **c1** that does not already exist in the **customer_num** column of the **customer** table.

A third kind of join, described under "LOOKUP" on page 2-48, allows you to display or verify data from a table that is not active.

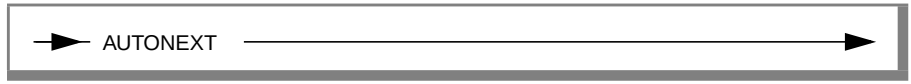
ATTRIBUTES Syntax

PERFORM recognizes the following attributes. The syntax for each attribute is detailed in the following sections.

```
AUTONEXT
CENTURY
COLOR
COMMENTS
DEFAULT
DOWNSHIFT
FORMAT
INCLUDE
INVISIBLE
LOOKUP
NOENTRY
NOUPDATE
PICTURE
PROGRAM
QUERYCLEAR
REQUIRE
REVERSE
RIGHT
UPSHIFT
VERIFY
WORDWRAP
ZEROFILL
```

AUTONEXT

Use the AUTONEXT attribute to cause the cursor to advance automatically to the next field when the current field is full.



Usage

- AUTONEXT is particularly useful for entering text into a CHAR type database column that is split among two or more display fields with the use of subscripts.
- Another use of AUTONEXT is with CHAR fields in which the input data is of a standard length (for example, the abbreviation for a state name is always two digits) or when the CHAR field has a length of one (only one keystroke is required to enter the data and to move to the next field).

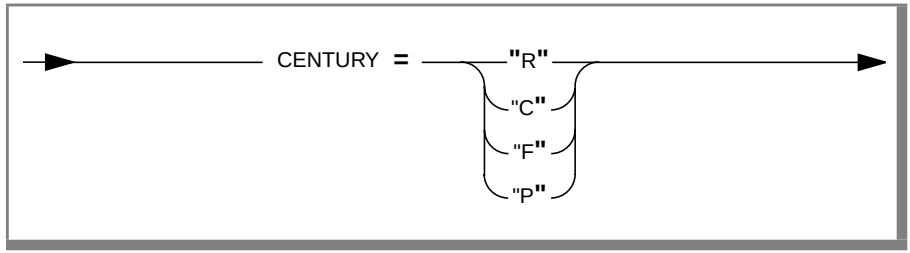
The **sample** form specification file uses the AUTONEXT attribute to display the **state** and **zipcode** columns from the **customer** table, as shown:

```
c8 = state, upshift, autonext;
c9 = zipcode, autonext;
```

When two characters are entered into the **c8** field (and the field is full), the cursor moves automatically to the beginning of the next field (the **c9** field). When five characters are entered into the **c9** field (and the field is full), the cursor moves automatically to the beginning of the next field.

CENTURY

The CENTURY attribute specifies how to expand abbreviated one- and two-digit *year* specifications in a DATE and DATETIME field. Expansion is based on this setting (and on the year value from the system clock at runtime).



C *or* c	is used for the past, future, or current year closest to the current date.
F *or* f	is used for the nearest year in the future to expand the entered value.
P *or* p	is used for the nearest year in the past to expand the entered value.
R *or* r	is used to prefix the entered value with the first two digits of the current year.

Usage

In most releases of INFORMIX-SQL earlier than 7.20, if the user enters only the two trailing digits of a year for literal DATE or DATETIME values, these are automatically prefixed with the digits *19*. For example, `12/31/02` is always expanded to `12/31/1902`, regardless of when the program is executed. This legacy behavior is sometimes called *the Y2K problem*.

CENTURY can specify any of four algorithms to expand abbreviated years into four-digit year values that end with the same digits (or digit) that the user entered. CENTURY supports the same settings as the **DBCENTURY** environment variable, but with a scope that is restricted to a single field.

Here *past*, *current*, and *future* are all relative to the system clock.

For more information on the **DBCENTURY** environment variable, see the *INFORMIX-SQL User Guide* and the *Informix Guide to SQL: Reference*.

Unlike **DBCENTURY**, which sets a global rule for expanding abbreviated year values in DATE and DATETIME fields that do not have the CENTURY attribute, CENTURY is not case-sensitive. You can substitute lowercase letters (`r`, `c`, `f`, `p`) for these uppercase letters. If you specify anything else (for example, a number), then `R` is used as the default. If the CENTURY and **DBCENTURY** settings are different, then CENTURY takes precedence.

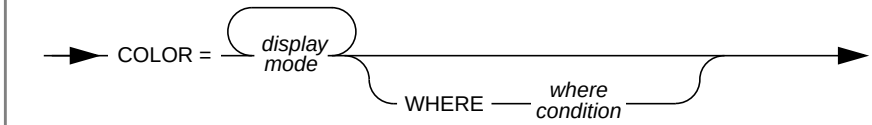
Three-digit years are not expanded. A single-digit year is first expanded to two digits by prefixing it with a zero. CENTURY then expands this value to four digits, according to the setting that you specified. Years between 99 BC (or BCE) and 99 AD (or CE) require leading zeros (to avoid expansion).

Just as with **DBCENTURY**, expansion of abbreviated years is sensitive to time of execution and to the accuracy of the system clock-calendar. To avoid the need to rely on CENTURY, require the user to enter four-digit years.

Important: *The* CENTURY *attribute has no effect on DATETIME fields that do not include YEAR as the first time unit, nor on fields that are not DATE or DATETIME fields. If an abbreviated year value is entered in a character field or a number field, for example, then neither* CENTURY *nor* **DBCENTURY** *has any effect.*

COLOR

Use COLOR to display field text in one of eight colors, either alone or combined with one or more of four intensities.



COLOR	is a required keyword.
display mode	is one or more display attributes selected from the list of available colors and intensities.
WHERE	is an optional keyword.
where condition	is a situation under which you want specified display attributes to be in effect.

Usage

The *display mode* consists of zero attributes or one attribute from the color list, and zero or more attributes from the intensity list, as follows.

Color	Intensity	Displayed As
WHITE		White
YELLOW		Yellow
MAGENTA		Magenta
RED		Red
CYAN		Cyan
GREEN		Green
BLUE		Blue
BLACK		Black

(1 of 2)

Color	Intensity	Displayed As
	BLINK	Blinking
	UNDERLINE	Underlined
	REVERSE	Reverse Video
	LEFT	Left Justified (number fields)

(2 of 2)

- If you do not select a *where condition*, the display mode always applies to the field. When you select a *where condition*, FORMBUILD tests the condition whenever a new value enters the field.
- If the condition is true, FORMBUILD displays the field with the mode you selected. If the condition is false, FORMBUILD displays the field with default characteristics.
- You can code the *where condition* syntax in any one of the following ways:

```
expr LIKE expr
expr NOT  LIKE expr
expr MATCHES expr
expr NOT  MATCHES expr
expr IS   NULL
expr IS  NOT  NULL
expr BETWEEN expr AND expr
expr NOT  BETWEEN expr AND expr
expr IN (list of exprs)
expr NOT  IN (list of exprs)
expr relop expr
(bool-expr)
bool-expr   OR  bool-expr
bool-expr  AND  bool-expr
```

where expr can represent any one of the following items:

```
field tag
constant
TODAY
CURRENT
agg-function OF field tag
- expr
expr [ + - * / ] expr
(expr)
```

and *relop* can be any of the following comparison operators:

```
=   <>   !=   >=   <=   <   >
```

The following example illustrates how to specify that field text should be displayed in red type:

```
f000 = customer.customer_num, color=red;
```

The following examples illustrate conditional use of COLOR:

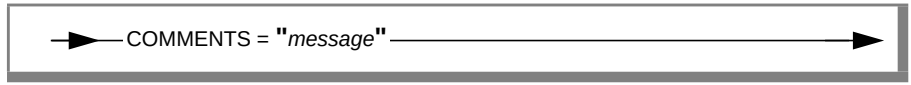
```
f002 = manufact.manu_code,
       color=red where f002="HRO";
f003 = customer.lname,
       color=red where f003 not like "Quinn";
f004 = customer.zipcode
       color=red blink where f004 > 10000;
```

GLS

The evaluation of MATCHES, LIKE, and BETWEEN expressions that contain character arguments is dependent on collation settings. Refer to Appendix C, "Global Language Support," and the *Informix Guide to GLS Functionality*.♦

COMMENTS

Use COMMENTS to cause PERFORM to display a message on the Comment line at the bottom of the screen. The message displays when the cursor moves to the associated field.



COMMENTS	is a required keyword.
message	is a character string enclosed in quotes.

Usage

- The *message* must appear in quotation marks on a single line of the form specification file.
- The Status line is the bottom line of the screen. The Comment line is just above the Status line.
- The most common use of the COMMENTS attribute is to give information or instructions to the user. This is particularly appropriate when the field accepts only a limited set of user-specified values.

An example from the **sample** form specification file follows:

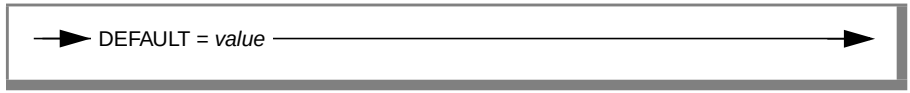
```
c2 = fname, comments =
   "Please enter initial if available." ;
```

Related Attribute

INCLUDE

DEFAULT

Use the DEFAULT attribute to assign a default value to a display field.



DEFAULT	is a required keyword.
value	is the default value.

Usage

- If you do not use the DEFAULT attribute, display fields default to blanks.
- Enclose DATE values and CHAR values that contain spaces or special characters in quotation marks. Using quotation marks around CHAR values that contain no spaces or special characters is optional.
- PERFORM displays the default value whenever the field displays for data entry in an Add operation.
- If both the DEFAULT attribute and the REQUIRED attribute are assigned to the same field, the REQUIRED attribute is ignored.
- Use the TODAY keyword as the *value* to assign the current date as the default value of a DATE field.
- Use the CURRENT keyword to assign the current date and time as the default value of a DATETIME field.
- If you use the WITHOUT NULL INPUT option in the DATABASE section and you do not use the DEFAULT attribute, then character fields default to blanks, number and INTERVAL fields default to `0`, and MONEY fields default to `$0.00`. The default DATE value is `12/31/1899`, and the default DATETIME value is `1899-12-31 23:59:59.99999`.
- If you do not use WITHOUT NULL INPUT in the DATABASE section, all fields default to NULL values unless you use the DEFAULT attribute.
- You cannot use DEFAULT with fields of type TEXT or BYTE.

Two examples from the **sample** form specification file follow:

```
c8  = state, upshift, autonext,
          default = "CA" ;

o12 = order_date, default = today,
          format = "mm/dd/yyyy" ;
```

DOWNSHIFT

Assign the DOWNSHIFT attribute to a CHAR field when you want PERFORM to convert uppercase letters to lowercase letters.

Usage

Because uppercase and lowercase letters have different ASCII values, storing character strings in one format or the other can simplify sorting and querying a database.

GLS

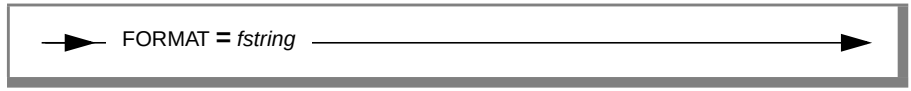
The results of conversion between uppercase and lowercase can be tailored to the national language in use, as defined by GLS settings. Refer to Appendix C, "Global Language Support," and the *Informix Guide to GLS Functionality*. ♦

Related Attribute

UPSHIFT

FORMAT

Use the FORMAT attribute with a DECIMAL, SMALLFLOAT, FLOAT, or DATE column to control the format of the display.



FORMAT	is a required keyword.
fstring	is a (format) string of characters that specifies the desired data format. You must enclose fstring in quotation marks.

Usage

- For DECIMAL, SMALLFLOAT, or FLOAT data types, *fstring* consists of pound signs (#) that represent digits and a decimal point. For example, ###.## produces up to three places to the left of the decimal point and exactly two places to the right.
- If the actual displayed number is shorter than the *fstring*, PERFORM right-justifies it and pads the left with blanks.
- If the *fstring* is smaller than the display width, FORMBUILD gives a warning, but the form is usable.
- If necessary, PERFORM rounds numbers before displaying them.

- For DATE data types, PERFORM recognizes the following symbols as special in the string *fstring*:

mm	produces the two-digit representation of the month.
mmm	produces a three-letter abbreviation of the month, for example, Jan, Feb, and so on.
dd	produces the two-digit representation of the day.
ddd	produces a three-letter abbreviation of the day of the week, for example, Mon, Tue, and so on.
yy	produces the two-digit representation of the year.
yyyy	produces a four-digit year.

  For dates, FORMBUILD interprets any other characters as literals and displays them wherever you place them within *fstring*.

- You cannot use the FORMAT attribute with DATETIME or INTERVAL data types.

The following table lists example FORMAT attributes for DATE fields.

Input	Result
no FORMAT attribute	09/15/1994
FORMAT = "mm/dd/yy"	09/15/94
FORMAT = "mmm dd, yyyy"	Sep 15, 1994
FORMAT = "yymmdd"	940915
FORMAT = "dd-mm-yy"	15-09-94
FORMAT = "(ddd.) mmm. dd, yyyy"	(Thu.) Sep. 15, 1994

Two examples from the **sample** form specification file follow:

```
o12 = order_date,
          default = today,
          format = "mm/dd/yyyy";

o22 = paid_date,
          format = "mm/dd/yyyy";
```

GLS

The way the format string in the FORMAT attribute is interpreted for numeric and monetary data can be modified by GLS settings. In the format string, the period symbol (.) is not a literal character but a placeholder for the decimal separator specified by environment variables. Likewise, the comma symbol (,) is a placeholder for the thousands separator specified by environment variables. The $ symbol is a placeholder for the leading currency symbol. The @ symbol is a placeholder for the trailing currency symbol. Thus, the format string *$$#,###.##* will format the value *1234.56* as *£1,234.56* in a British locale but as *f1.234,56* in a French locale. Refer to Appendix C, "Global Language Support," and the *Informix Guide to GLS Functionality*.

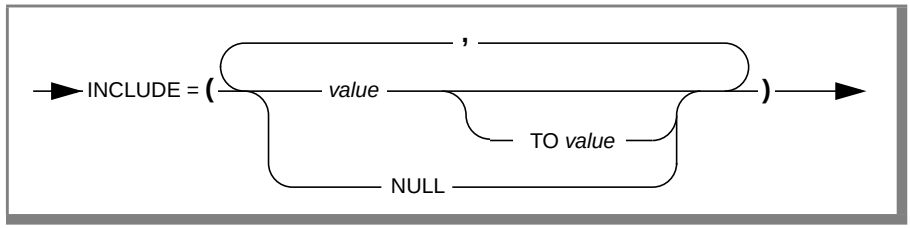
The *mmm* and *ddd* specifiers in a format string can display language-specific month name and day name abbreviations on the form. This requires the installation of message files in a subdirectory of **$INFORMIXDIR/msg** and subsequent reference to that subdirectory by way of the environment variable **DBLANG**. For example, the *ddd* specifier in a Spanish locale translates the day Saturday into the day name abbreviation *Sab*, which stands for "Sabado" (the Spanish word for Saturday). ♦

Related Attribute

PICTURE

INCLUDE

Use the INCLUDE attribute to specify acceptable values for a field and to cause PERFORM to check input before accepting it.



INCLUDE	is a required keyword.
value	is either a list of individual values (value1, *value2*, ...), a range of values (*value1 TO value2*), or a combination of individual values and ranges, all separated by commas.
NULL	is an optional keyword that specifies a NULL value.

Usage

- When you specify a range of values, the lower value must appear first.
- For ranges of character values, PERFORM uses dictionary ordering with the printable ASCII character set. (See Appendix E, "The ASCII Character Set," for an ordered list of the ASCII character set.) In a number field, the range (5 to 10) is acceptable. In a CHAR field, it is incorrect. The character string 10 is less than the string 5, since 1 comes before 5 in the ASCII character set.
- If you include a character string that contains a blank space, a comma, or any special characters, you must enclose the entire string in quotation marks. It is advisable to enclose character strings in quotation marks at all times.
- Before PERFORM accepts a new row, you must enter an acceptable value in each display field with the INCLUDE attribute. Use the keyword NULL to indicate that a NULL value is acceptable. If a field has both the DEFAULT and INCLUDE attributes, then the DEFAULT value must appear in the INCLUDE list. Otherwise the form does not compile.

- Including COMMENTS that indicate acceptable values makes data entry easier.

An example from the **sample** form specification file follows:

```
i18 = items.quantity,
      include = (1 to 50),
      comments = "Acceptable values are 1 through 50";
```

GLS

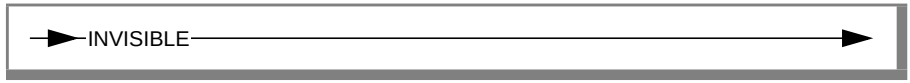
The results of evaluation of character data in INCLUDE ranges can be affected by GLS settings. A given character will be contained in an INCLUDE range or not depending on where it collates relative to the INCLUDE values. Refer to Appendix C, "Global Language Support," and the *Informix Guide to GLS Functionality*. ♦

Related Attributes

COMMENTS, REQUIRE

INVISIBLE

If a field is defined as INVISIBLE, INFORMIX-SQL does not display the value assigned to the field or the value the user is entering in the field.

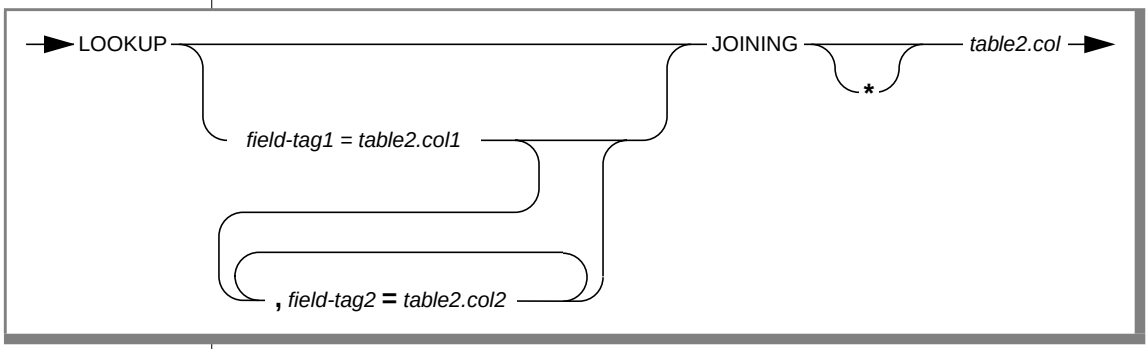


Usage

- If you assign both INVISIBLE and COLOR attributes to a field, INFORMIX-SQL ignores the COLOR attribute, unless you specify COLOR=REVERSE. In this case, INFORMIX-SQL displays the field in reverse video and maintains the invisibility of the field's contents.
- If you assign both INVISIBLE and PICTURE attributes to a field, INFORMIX-SQL does not display the picture pattern.

LOOKUP

Use the LOOKUP attribute to display data from another table while entering data into or querying the active table. You can also use it to prevent data from being entered into the active table if the value does not exist in another table.



LOOKUP	is the keyword that specifies a join.
field-tag1	is the field tag of a field that displays a value from the LOOKUP table.
table2.col1	is a column in table2 whose value displays in *field-tag1*.
JOINING	is the keyword that identifies the joined column.
*	is optional punctuation that identifies the dominant column in a verify join. (See "Verify Joins" on page 2-30.)
table2.col	is the name of a column that belongs to table2 and is joined to *table1.col*.
field-tag2	is the field tag of a field that displays a value from the LOOKUP table.
table2.col2	is a column in table2 whose value displays in *field-tag2*.

Usage

- If you use an alias in the TABLES section, you must use the alias to refer to the table in the ATTRIBUTES section.
- The optional asterisk placed in front of *table2.col* tells PERFORM to accept a value for *table1.col* only if the same value already exists in *table2.col*.

- The optional list of field tags with column names following the LOOKUP attribute directs PERFORM to display these values whenever there is a successful join between *table1.col* and *table2.col*. You cannot enter values into these fields from the keyboard.
- If the join columns in a LOOKUP are not indexed, the LOOKUP does not run as quickly.

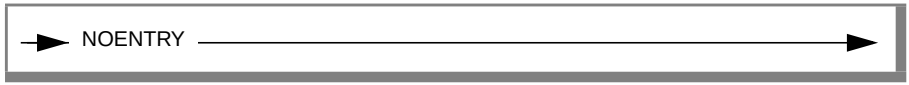
An example of the LOOKUP join from the **sample** form specification file follows:

```
i16 = items.manu_code,
       lookup m17 = manufact.manu_name
       joining  *manufact.manu_code;
```

In this example, the entry of the item manufacturer code number is checked against the list of manufacturer code numbers in the **manufact** table. If the same value is found there, the manufacturer's name is extracted from the **manufact** table and displays in field **m17**.

NOENTRY

Use the NOENTRY attribute to prevent data entry when a new row is created during an Add operation.



Usage

- The NOENTRY attribute does not prevent you from modifying the field during an Update operation.
- The NOENTRY attribute is unnecessary with a SERIAL column.

Two examples from the **sample** form specification file follow:

```
i13 = items.stock_num;
    = *stock.stock_num, noentry,
        noupdate, queryclear;

s14 = stock.description, noentry,
        noupdate;
```

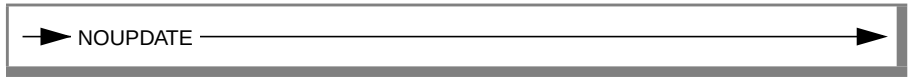
When the **stock** table is active, the columns **i13** and **s14** (corresponding to the columns **stock.stock_num** and **stock.description**, respectively) cannot have values added. (The inclusion of the NOUPDATE attribute prevents data entry during an Update operation.)

Related Attribute

NOUPDATE

NOUPDATE

Use the NOUPDATE attribute to prevent data entry when a row is modified during an Update operation.



Usage

The NOUPDATE attribute does not prevent you from entering data into the field during an Add operation.

Two examples from the **sample** form specification file follow:

```
s15 = stock.unit_price, noentry,
          noupdate;

s16 = stock.unit_descr, noentry,
          noupdate;
```

When the **stock** table is active, the fields **s15** and **s16** (corresponding to the columns **stock.unit_price** and **stock.unit_descr**, respectively) cannot receive values during an Update operation. (The inclusion of the NOENTRY attribute prevents data entry during an Add operation.)

Related Attribute

NOENTRY

PICTURE

Use the PICTURE attribute to specify the character pattern for data entry to a non-number field.

PICTURE	is a required keyword.
pstring	is a (picture) string of characters that specifies the desired character pattern.

Usage

- *pstring* is a combination of three special symbols.

Symbol	Meaning
A	Any letter
#	Any digit
X	Any character

  Any other character in the *pstring* is treated as a literal and occurs, during data entry, in the exact location indicated.

- If you attempt to enter a character that does not conform with the *pstring*, you hear a beep and PERFORM does not echo the character on the screen.
- The PICTURE attribute does not require the entry of the entire field; it only requires that what you enter conforms to *pstring*. Note that the length of *pstring* must equal the length of the corresponding display field.
- PERFORM reminds data entry operators of the required pattern by displaying the literal characters in the display field and leaving blanks elsewhere.

The following examples are from the **sample** form specification file:

```
c10 = phone,
          picture = "###-###-####x####";
```

produces the following display field before data entry:

```
[   -   -          ]
```

As another example, if you specify a field for part numbers like this:

```
f1 = part_no, picture = "AA#####-AA(X)";
```

PERFORM would accept any of the following inputs:

```
LF49367-BB(*)
TG38524-AS(3)
YG67491-ZZ(D)
```

GLS

The PICTURE attribute is not affected by the GLS settings because PICTURE only formats character information. ♦

Related Attribute

FORMAT

PROGRAM

You can use the PROGRAM attribute with a blob (BYTE or TEXT) column to call an external program to work with the TEXT or BYTE data. You invoke an external program by pressing the exclamation key while your cursor is in a blob field. The external program then takes over control of the screen. When you exit the external program, the form is restored on your screen.

The syntax of the PROGRAM attribute follows.

field-tag	is the field tag used in the SCREEN section.
table.col	is the name of a field, either related to a column, form-only field, or display-only (INFORMIX-SQL) field.
PROGRAM	is a required keyword.
name	is a command string to, or the name of, a batch file that invokes an editing program.

Usage

If you call the program on an empty field, when you finish working in the external program and save your work, the data is stored in the blob field. If you call the program from a field that already contains data, the specified program works on the data in that field. In either case, Informix Dynamic Server writes the blob to a temporary file, which is then passed to the external program. The external program must write its changes back to the temporary file. You do not need to know the name of the temporary file; the application development tool keeps track of it. For example, you might use PROGRAM to call a CAD or graphics program to display a drawing that you have stored. You can also use PROGRAM to invoke an editor for a TEXT field.

For example, a TEXT field might be tagged with the following line:

```
f003 = personnel.resume, WORDWRAP, PROGRAM = "edit";
```

When you display a field with data type TEXT, INFORMIX-SQL displays as many of the leading characters as will fit in the defined field. When you display a field with data type BYTE, INFORMIX-SQL displays <BYTE value>.

When you place the cursor in a TEXT field, and you press the exclamation-mark key in the first character position of a TEXT or BYTE field, the external program is invoked. This program receives the contents of the field and takes control of the screen to permit editing or alteration of the field. When the program is finished, your application regains control of the screen and continues execution.

One of the following programs is invoked for a TEXT field:

- The program specified by the PROGRAM ="name" attribute defined for that field, if any.
- The program named in the **DBEDIT** environment variable, if one is defined.
- The default editor, which depends on the host operating system.

You must explicitly define the external program for a BYTE field; the default editor is not called, and the **DBEDIT** environment variable is not examined.

Before invoking the program, INFORMIX-SQL copies the BYTE or TEXT field to a temporary disk file. It then issues a system command composed of the name that you specify after the PROGRAM keyword followed by the name of the temporary file.

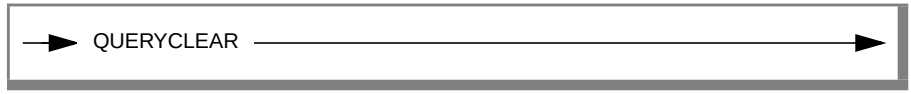
The name string need not be a single word. You can add additional command parameters. The program can also be a shell script, so that you can initiate a whole series of actions.

If you are invoking an external program from PERFORM, the data is stored in the blob column when you complete the Add or Update. For example:

```
f010  = contract, PROGRAM = "edit";
```

QUERYCLEAR

Use the QUERYCLEAR attribute to clear a joining field on the screen when you enter a Query operation.



Usage

- When you enter the Query option, PERFORM normally clears all fields except joining and display-only fields.
- QUERYCLEAR does not apply to display-only fields. You must give explicit instructions in the INSTRUCTIONS section to clear display-only fields.

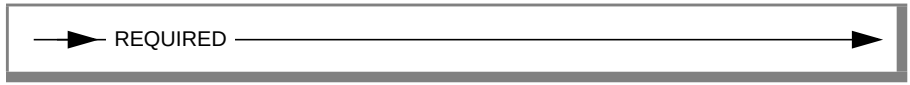
An example from the **sample** form specification file follows:

```
i13 = items.stock_num;
    = *stock.stock_num, noentry,
          noupdate, queryclear;
```

Here the **items** table and the **stock** table are joined through the stock number. When the **stock** table is the active table and a query is made, the **stock_num** field is cleared. When **items** is the active table, however, the **stock_num** field is not cleared when a query is made.

REQUIRED

Use the REQUIRED attribute to force data entry into a particular field during an Add operation.



Usage

- The REQUIRED attribute has no effect during a PERFORM Update operation. You are free to erase values from REQUIRED fields when you use an Update operation.
- There is no default value for a REQUIRED field. If you assign both the REQUIRED attribute and the DEFAULT attribute to the same field, the REQUIRED attribute is ignored.

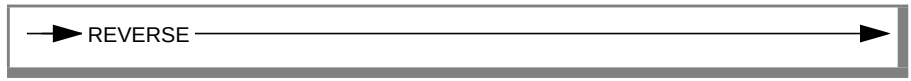
An example from the **sample** form follows:

```
o20 = po_num, required;
```

FORMBUILD requires the entry of a purchase order value when adding a new order to the database.

REVERSE

Assign the REVERSE attribute to fields you want PERFORM to display in reverse video.

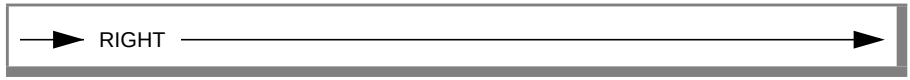


Usage

On computers that do not support reverse video, fields that have the REVERSE attribute are enclosed in angle brackets (< >).

RIGHT

Assign the RIGHT attribute to fields in which you want the data to be right-justified.

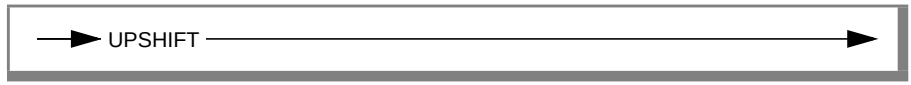


Usage

- PERFORM right-justifies data you enter during an Add or Update operation.
- To search for a right-adjusted CHAR field of value "*string*" during a Query operation, use the wildcard search pattern "**string*" to account for potential leading blanks.

UPSHIFT

Assign the UPSHIFT attribute to a CHAR field when you want PERFORM to convert lowercase letters to uppercase letters.



Usage

Because uppercase letters and lowercase letters have different ASCII values, storing character strings in one format or the other can simplify sorting and querying of a database.

An example from the **sample** form follows:

```
c8  = state, upshift, autonext,
          include = ("CA", "OR", "NV", "WA"),
          default = "CA" ;
```

Because of the UPSHIFT attribute, PERFORM enters uppercase characters in the **state** field regardless of the case used to enter them.

GLS

The results of conversion between uppercase and lowercase can be made appropriate for different languages using GLS settings. Refer to Appendix C, "Global Language Support," and the *Informix Guide to GLS Functionality*. ♦

Related Attribute

DOWNSHIFT

VERIFY

Use the VERIFY attribute when you want PERFORM to require users to enter data twice for a particular field to reduce the probability of erroneous data entry.

Usage

Because some data is critical, the VERIFY attribute supplies an additional step in data entry to ensure the integrity of the data in your database. After you enter a value into a VERIFY field and press RETURN, PERFORM erases the field and requests that you reenter the value. You must enter *exactly* the same data each time, character for character: `15000` is not exactly the same as `15000.00`.

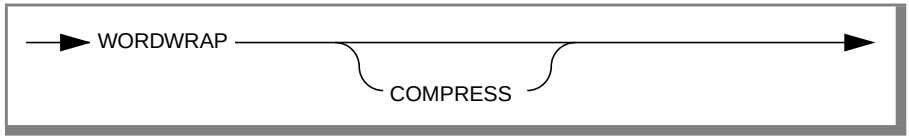
If you specify the following field for salary information:

```
s10 = salary, verify;
```

PERFORM requires the entry of exactly the same data twice.

WORDWRAP

Use the WORDWRAP attribute when you want PERFORM to wrap a long character string to the next field that has the same field tag.



WORDWRAP	is the required keyword that instructs PERFORM to wrap long character strings to the next successive field.
COMPRESS	is the optional keyword that tells PERFORM to discard any spaces that you did not type and that are not part of the data.

Usage

- The keyword WORDWRAP enables the multiline editor. When you enter text from the keyboard and reach the end of a line, the editor brings the current word down to the next line, moving text to subsequent lines as necessary. When you delete text, the editor pulls words up from lower lines whenever it can.

  If you do not use the WORDWRAP attribute, words do not flow from one line in the field to the next, and you must edit text by using the arrow keys or the RETURN key to move from field to field.
- The editor distinguishes between *intentional* blanks (blanks that you typed or that are part of the data) and *editor* blanks (blanks that the editor inserts at the ends of lines to make text wrap around to the next line). Intentional blanks are retained as part of the data. Editor blanks are inserted and deleted automatically as required for word wrapping.
- The COMPRESS attribute tells PERFORM to discard editor blanks. If you do not use the COMPRESS attribute, and the sum of the segment lengths exceeds the column size, PERFORM might truncate some trailing words.
- When you design a multiline field, allow room for editor blanks. You can expect the average number of editor blanks per line to be half the length of an average word.

- The editor breaks lines between words whenever possible. Ordinarily, the field is as long as, or longer than, the column size, and PERFORM displays all text.
- If the column data is longer than the field, the editor fills the field and discards the excess data. You lose data if you use a truncated display to update a database.

The following example shows the SCREEN and ATTRIBUTES sections of a form specification file that specifies a multiple-line field:

```
database...
screen
{
Enter text:   [mlf                    ]
              [mlf                    ]
              [mlf                    ]
              [mlf                    ]
}
tables...
attributes
   mlf = charcolm, wordwrap compress;
```

Because the screen field whose tag is **mlf** appears in four physical segments in the screen layout and has the WORDWRAP attribute, it is a multiple-line field. Its value is composed of the physical segments taken in top-to-bottom, left-to-right order. The field should ordinarily be as long as or longer than the column so that it can display all of the text. It is not necessary that the segments be the same size, as they are in the example.

In the field description in the ATTRIBUTES section, the keyword WORDWRAP enables the multiline editor. If you omit it, words cannot flow from one segment to the next.

If a field is defined to accept VARCHAR data, you must assign the WORDWRAP attribute to the field to enable the multiline editor.

VARCHARs are similar to character fields; both are supported by the multiline editor. You must assign the WORDWRAP attribute to VARCHAR fields to enable the multiline editor. For example, the following excerpt from a form specification shows the VARCHAR field **history** in the **employee** table and the attributes assigned to the field:

```
history     [f002                                          ]
            [f002                                          ]
            [f002                                          ]

attributes

f002 = employee.history, WORDWRAP COMPRESS;
```

If you generate a default form for a table that has a VARCHAR column, the VARCHAR field is broken into subscripted fields. To enable WORDWRAP, revise the form and use the same field tag for all the components of the VARCHAR field; then add the WORDWRAP and COMPRESS attributes.

You can use VARCHAR as the data type for a display-only (FORMBUILD) field.

You can use the DEFAULT attribute to give a VARCHAR field a default value.

Specifying TEXT and BYTE Data

You can use columns of types TEXT or BYTE as fields. Assign the WORDWRAP attribute to a TEXT field to display the field so that it fits into the display without having any lines start with a blank. For a TEXT field, the WORDWRAP attribute only affects how the value is displayed; WORDWRAP does not enable the multiline editor. If you want to edit a TEXT field, you must use the PROGRAM attribute to indicate the name of an external editor.

ZEROFILL

Assign the ZEROFILL attribute to fields that you want to be right-justified and padded with leading zeros.

Usage

This attribute is most useful with numeric fields. If the number entered into the field is shorter than the field itself, PERFORM right-justifies it and fills the leading blanks with zeros.

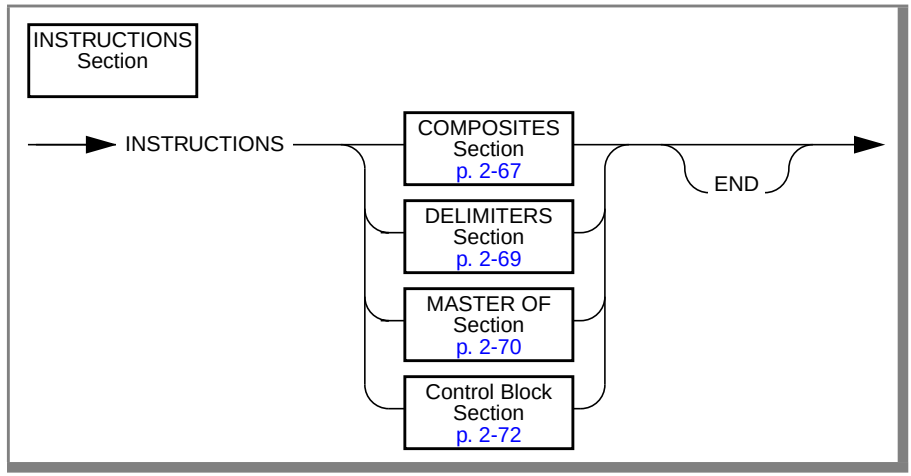
INSTRUCTIONS Section

The final section of the form specification file is the optional INSTRUCTIONS section. This section is used for the following tasks:

- Establishing composite joins
- Specifying alternative field delimiters
- Creating master / detail relationships
- Defining control blocks

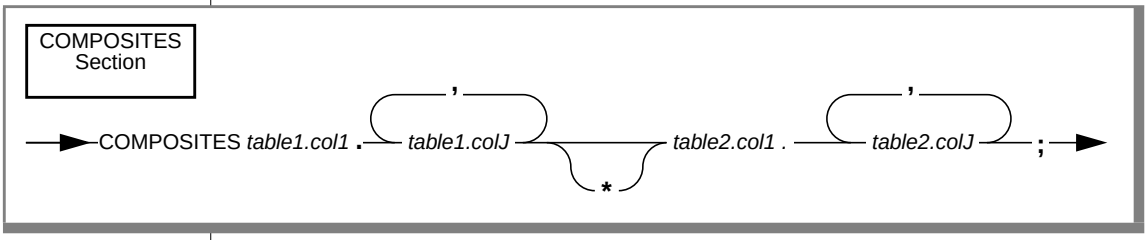
You can also call C functions from within the INSTRUCTIONS section. For details, see Chapter 6, "Functions in ACE and PERFORM."

The INSTRUCTIONS keyword begins the INSTRUCTIONS section as shown in the following diagram.



COMPOSITES

Establish a COMPOSITE JOIN between two tables when you must specify the values of more than one column in a table to specify a row uniquely.



COMPOSITES	is the keyword indicating that the following sets of column names enclosed in angle brackets (< >) are to be treated as composite columns that are joined to each other.
table1.colJ	(where J = 1, 2, 3, ...) is a column in table1.
table2.colJ	(where J = 1, 2, 3, ...) is a column in table2.
*	indicates that the join is a verify join—that is, unless the marked composite exists in *table2*, PERFORM does not allow the corresponding row to be written to *table1*.

Usage

- If you use an alias in the TABLES section, you must use that alias to refer to the table in the composite join.
- Each column included in a composite join must also be individually joined in the ATTRIBUTES section of the form specification. This means that *table1.col1* must be joined individually to *table2.col1* in the ATTRIBUTES section, as must *table1.col2* to *table2.col2*, and so on.
- There can be no additional joins between columns of the two tables that are not included in the composite join.
- If the columns in a composite join are not individually and jointly indexed, cross-table queries do not run as quickly.

An example from the **sample** form specification file follows:

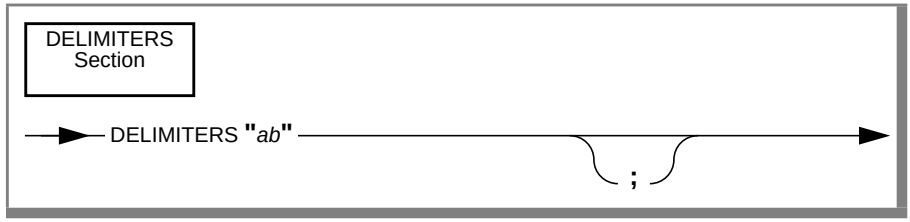
```
composites <items.stock_num, items.manu_code>
       * <stock.stock_num, stock.manu_code>
```

The **stock_num** and **manu_code** fields in the **items** and **stock** table are included in a composite join. This is a composite verify join. When the **items** table is active, values entered in the **stock_num** and **manu_code** fields are compared with values existing in those two columns in the **stock** table. PERFORM notifies the user if there is not a match and rejects the entry. This precludes the entry of stock numbers and manufacturer codes that individually exist in the database but, as a composite, do not correspond to a unique row in the **stock** table.

To specify a unique row in the **stock** table requires both the **stock_num** and **manu_code**. For example, the **stock** table contains three rows with the stock number 1, and four rows with the manufacturer code HRO. (See Appendix A, "The Demonstration Database and Examples," for a list of data included in the sample database.) Knowing the stock number or manufacturer code alone does not allow you to locate a unique row. You need both the stock number (1) and the manufacturer code (HRO) to specify a unique row (baseball gloves produced by Hero) in the table.

DELIMITERS

You can change the delimiters that PERFORM uses to enclose the fields when the form appears on the screen. The default delimiters are brackets ([ ]), but you can substitute any other printable character, including blank spaces.



DELIMITERS	is a required keyword.
a	is the opening delimiter.
b	is the closing delimiter.

Usage

- The DELIMITERS instruction tells PERFORM the symbol to use as a delimiter when it displays the fields on the screen.
- Each delimiter is a single character only.
- FORMBUILD still requires that you use brackets in the form specification file.
- If your form has columns from more than one database table, you might not want to use blank spaces as delimiters. If you use blank spaces, you have no visual indication on the screen of which fields correspond to columns in the active table.
- You can use the | bar symbol to denote both a closing delimiter and an opening delimiter. For example,

```
Name   [tag1        |tag2         ]
```

`tag1` identifies the first display field; `tag2` identifies the second display field. If you use the bar symbol in the SCREEN section, you must include a DELIMITERS statement in the INSTRUCTIONS section of the form specification. Use two identical symbols for the left and right delimiter in the DELIMITERS statement.

MASTER OF

Create a master / detail relationship between two tables when a row of one table (master) is associated with several rows of another table (detail).

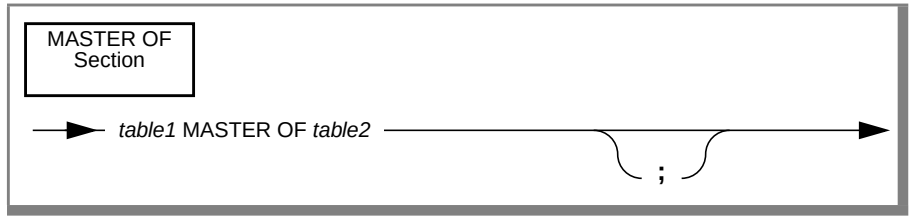


table1	is a table in the database that is designated as the master table.
MASTER OF	are required keywords.
table2	is a table in the database that is designated as the detail table.

Usage

- If you have used an alias in the TABLES section, you must use that alias to refer to the table in the master / detail relationship.
- You cannot include a temporary table in your table list.
- The master / detail relationship simplifies cross-table queries, especially when one row of *table1* is associated with several rows of *table2*.
- Master / detail relationships can be defined in both directions.
- If no explicit master / detail relationship exists, PERFORM displays an error message when you use the Master or Detail option.

Two examples from the **sample** form specification file follow:

```
customer master of orders;
orders master of items;
```

These master / detail relationships are useful because each customer can have many orders, and each order can have many items.

Additional examples are displayed in the following table.

Master	Detail
projects	personnel
orders	items
agents	clients
parents	children

These master / detail relationships are useful where several staff members (**personnel**) work on the same project (**projects**), each purchase order (**orders**) contains more than one item (**items**), or a single agent (**agents**) has many clients (**clients**).

With a master / detail relationship defined in both directions, you can explore the database in the following way. Suppose you have a database that consists of **personnel** and **projects** tables. Each person is assigned to a single project, and each project has several people working on it. The screen form includes an INSTRUCTIONS section stating:

```
personnel master of projects;
projects master of personnel;
```

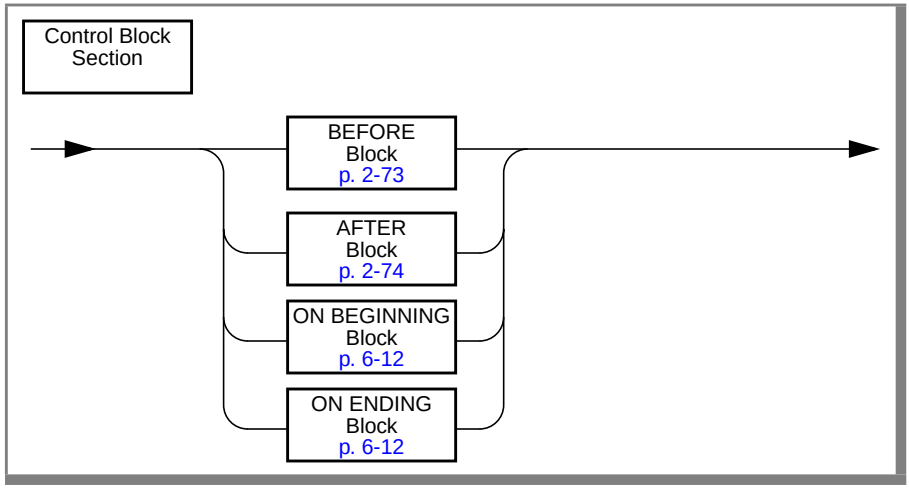
Assume that you want to query the database to find all employees who work with a particular employee, but you do not know on which project they work. When you identify and bring the particular employee to the screen (the **personnel** table is active) and select the Detail option, PERFORM moves to the PROJECT INFORMATION screen (the **projects** table is active) and displays the information on the employee's project. If you then choose the Detail option, PERFORM selects all employees on that project and shifts to the PERSONNEL INFORMATION screen (the **personnel** table is active).

Control Blocks

Use control blocks to perform these functions:

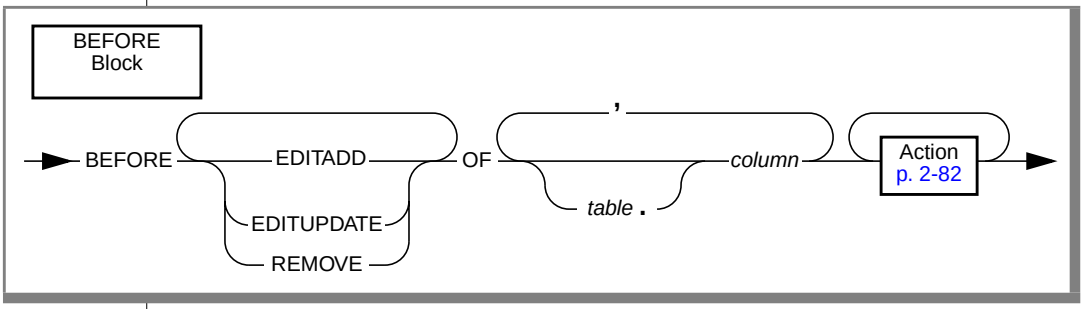
- Control the cursor movement when you add or update a row.
- Check the value of data you enter against criteria that depend on other data that has already been entered.
- Modify the data in fields after Add, Update, and Query operations.
- Perform calculations on field values and enter the results into another field.
- Display aggregate information like averages and totals on columns in the current list. (The current list is the set of rows that results from a Query as modified by subsequent Add or Remove actions.)
- Call C functions from PERFORM. For details, see Chapter 6.

Each control block is either a BEFORE block or an AFTER block. Screen control actions can be taken either before or after PERFORM operations are completed. You can use BEFORE blocks with the Add, Update, and Remove operations. You can use AFTER blocks with the Add, Update, Query, Remove, and Display operations.



BEFORE

Use a BEFORE control block to cause PERFORM to take a series of actions before it executes an operation.

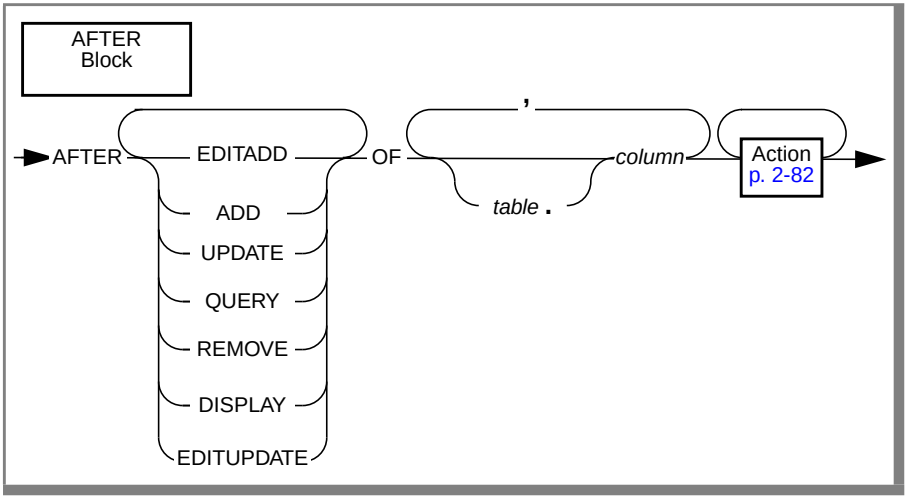


BEFORE	is a required keyword.
OF	is a required keyword.
table.column	is a list of up to 16 names or aliases of database tables and / or names of columns. Depending on your operating system, the limit on the number of table names may be lower.

- The EDITADD and EDITUPDATE keywords refer to the act of editing during an Add and an Update, respectively. For EDITADD and EDITUPDATE, the actions are taken before PERFORM writes the row to the table.
- You can use the BEFORE REMOVE operation with the ABORT keyword (described on page 2-83) to prevent a user from removing the last row from a detail table.

AFTER

Use an AFTER control block to cause PERFORM to take a series of actions after it executes an option.

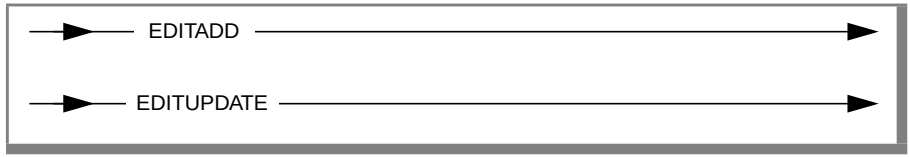


AFTER	is a required keyword.
OF	is a required keyword.
table.column	is a list of up to 16 names or aliases of database tables and / or names of columns. Depending on your operating system, the limit on the number of table names may be lower.

- The ADD, UPDATE, QUERY, and REMOVE keywords correspond to the PERFORM options Add, Update, Query, and Remove, respectively. For more information about the PERFORM options, see Chapter 3, "The PERFORM Screen Transaction Processor,".
- The DISPLAY keyword refers to the display of fields after PERFORM executes Next, Previous, Query, or other options.
- EDITADD and EDITUPDATE differ from ADD and UPDATE. For EDITADD and EDITUPDATE, the actions are taken before PERFORM writes the row to a table. For ADD and UPDATE, they are taken after PERFORM writes the row to the table.
- You can list only table names or aliases, including **displaytable** in *table.column* following the ADD, UPDATE, QUERY, REMOVE, and DISPLAY keywords.

EDITADD and EDITUPDATE

The EDITADD and EDITUPDATE keywords give you the ability to perform one or more actions before or after you enter data into a field during an Add and an Update operation, respectively. The action occurs before the row is written to the table.



Usage

- If you are using EDITADD or EDITUPDATE in a BEFORE control block and the *table.column* contains the names of columns only, the BEFORE keyword instructs PERFORM to execute the actions when the cursor moves to the corresponding field, before you enter data.
- If you are using EDITADD or EDITUPDATE in an AFTER control block and the *table.column* contains the names of columns only, the AFTER keyword instructs PERFORM to execute the actions when you enter data into the corresponding field and press RETURN. PERFORM makes all the attribute-specified checks (such as INCLUDE, VERIFY, and so on) before executing the actions.
- When you specify a database table or alias instead of a column in a BEFORE block, PERFORM executes the actions before you enter any data into the form. Using this feature, you can make PERFORM enter defaults into fields and display comments depending on the active table.
- When you specify a database table or alias instead of a column in an AFTER block, PERFORM executes the actions after you enter all the data and press ESCAPE to complete the transaction, before the row is written to the database. Using this technique, you can make consistency checks of all the data entered and return to data entry if you find inconsistencies.

- In a BEFORE block, when you refer to a CHAR column that is split into more than one field, PERFORM executes the actions before each section of the displayed field. If you want these actions executed only before the first section of a split field, replace the BEFORE block of the split field with an AFTER block of the immediately preceding field.
- If you want the actions executed only after the last section of a split field, replace the AFTER block of the split field with a BEFORE block of the immediately succeeding field.

The following examples are taken from the **sample** form specification file at the end of this chapter. The syntax of the action statements used in these examples is described in "Action Syntax" on page 2-82.

```
after editadd editupdate of quantity
   let i19 = i18 * s15
   nextfield = o11
```

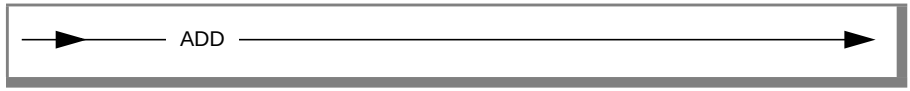
After you enter a value in the **quantity** field (using the Add or Update options), PERFORM calculates and places the value in the **i19** (Total Price) column, and places the cursor in the **o11** (Order Number) field.

```
before editadd editupdate of orders
   nextfield = o20
```

In this example, as soon as you indicate that you want either the Add or Update options when **orders** is the active table, PERFORM is instructed to move the cursor to the **o20** (Customer P.O.) field. Without this instruction, the cursor would go first to the **o11** (Order Number) field because it is the first **orders** field to appear in the ATTRIBUTES section of the form.

ADD

Use the ADD keyword to cause PERFORM to execute actions after the Add operation. The action occurs after the row is written to the table.



Usage

The main use of the ADD keyword involves keeping track of the number of rows written, computing statistics on the values entered into particular fields, and other bookkeeping operations.

The following example is from the **sample** form. The action statements used in this example are described in "Action Syntax" on page 2-82.

```
after add update query of items
   if (total of i19) <= 100 then
      let d1 = 7.50
      else
      let d1 = (total of i19) * .04

   let d2 = (total of i19) + d1
```

After you press ESCAPE following an Add, Update, or Query of the **items** table, PERFORM calculates values for the **d1** and **d2** fields and displays the values on the screen.

UPDATE

Use the UPDATE keyword to cause PERFORM to execute actions after the Update operation.

Usage

The main use of the UPDATE keyword involves keeping track of the number of rows written, computing statistics on the values entered into particular fields, and other bookkeeping operations.

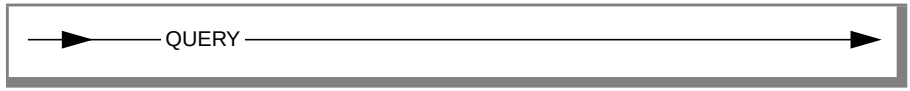
The following example is from the **sample** form specification file. The action statements used in this example are described in "Action Syntax" on page 2-82.

```
after add update query of items
   if (total of i19) <= 100 then
      let d1 = 7.50
      else
      let d1 = (total of i19) * .04

   let d2 = (total of i19) + d1
```

After you press ESCAPE following an Add, Update, or Query of the **items** table, PERFORM calculates values for the **d1** and **d2** fields and displays the values on the screen.

QUERY

Use the QUERY keyword to cause PERFORM to execute actions after the Query operation.



Usage

The main use of the QUERY keyword involves keeping track of the number of rows written, computing statistics on the values entered into particular fields, and other bookkeeping operations.

The following example is from the **sample** form specification file. The action statements used in this example are described in "Action Syntax" on page 2-82.

```
after add update query of items
   if (total of i19) <= 100 then
      let d1 = 7.50
      else
      let d1 = (total of i19) * .04

   let d2 = (total of i19) + d1
```

After you press ESCAPE following an Add, Update, or Query of the **items** table, PERFORM calculates values for the **d1** and **d2** fields and displays the values on the screen.

REMOVE

Use the REMOVE keyword to cause PERFORM to execute actions before or after the Remove operation.

Usage

- The main use of the AFTER REMOVE operation involves keeping track of the number of rows removed, computing statistics on the values entered into particular fields, and other bookkeeping operations.
- Use the BEFORE REMOVE operation to cause PERFORM to take one or more actions before removing a row from a database table.

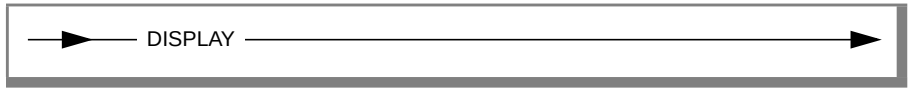
The following statement prints a message on the screen whenever a user selects the Remove option:

```
. . .

instructions

before remove of customer
   comments reverse
   "Remember to send a notice to the sales department"
. . .
```

You can use the BEFORE REMOVE operation with the ABORT keyword (see "ABORT" on page 2-83) to prevent a user from removing the last row from a detail table.

DISPLAY

Use the DISPLAY keyword to cause PERFORM to execute actions after any of the PERFORM operations that cause data to be displayed on the screen.



Usage

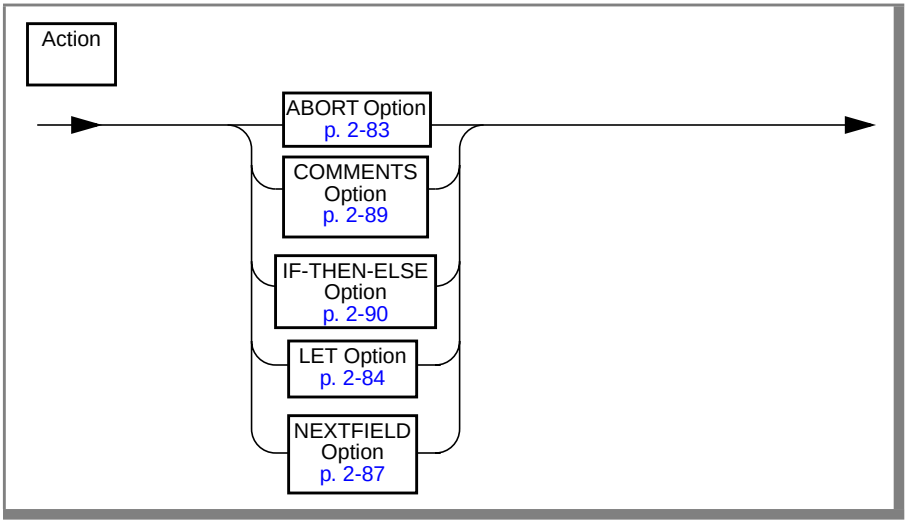
The following example is taken from the **sample** form specification file at the end of this chapter.

```
after display of orders
   let d1 = 0
   let d2 = 0
```

As soon as the data displays when the **orders** table is active, this control block instructs PERFORM to set the values in the **d1** (Ship Charge) and **d2** (Total Order Amount) fields to zero.

Action Syntax

This section provides the syntax of the following actions.

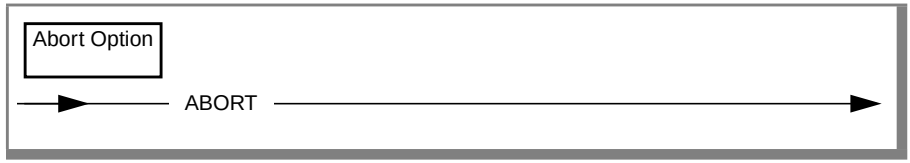


ABORT	exits to the PERFORM menu without making a change to the database.
COMMENTS	displays a message on the Status line.
IF-THEN-ELSE	performs other actions based on conditions among the values in the fields.
LET	assigns values to fields.
NEXTFIELD	moves the cursor to a specific field or exits to the PERFORM menu.

For these actions to compile properly, you must include them in a BEFORE or AFTER control block.

ABORT

Use the ABORT keyword in the INSTRUCTIONS section of a form specification to end a current Add, Update, or Remove action without altering the database and return to the PERFORM menu.



Usage

- The ABORT action compares to the NEXTFIELD EXITNOW action in the following respects:
  - With NEXTFIELD EXITNOW, PERFORM executes an Update, Remove, or Add, and then exits to the PERFORM menu. NEXTFIELD EXITNOW is equivalent to pressing ESCAPE.
  - With ABORT, PERFORM exits to the PERFORM menu without executing an Update, Remove, or Add. ABORT is equivalent to pressing CONTROL-C.
- You can use the ABORT keyword with the EDITADD, EDITUPDATE, and REMOVE options.

  For example, suppose you maintain a master table with employee information and a detail table with information about employee projects (joined to the master table by employee number). Projects are added and deleted on a regular basis, and you want to ensure that all employees have projects. (It is an administrative or clerical error to remove the last detail row, thereby leaving the employee with no project.) You can use the ABORT keyword with the BEFORE REMOVE operation to call a C function that checks the number of rows in the detail table. If the current row is the last detail row, the operation aborts. For information about calling C functions from PERFORM, see Chapter 6, "Functions in ACE and PERFORM."

LET

Use the LET action to attach a value to a field tag for display on the form.

LET	is a required keyword.
field-tag	is the field tag of a display-only field, of a column named in the *table.column* list in the control block, or of a column belonging to each of the tables named in the *table.column* list.
expression	is an expression as defined below.

Usage

- FORMBUILD gives an error if *field-tag* does not satisfy the preceding conditions.
- You can assign values only to fields corresponding to columns in the active table or to display-only fields.
- An expression is:
  - A field tag
  - A constant value
  - One of the aggregate functions followed by the phrase OF *tagname*, where *tagname* is the field tag of a database column and not the name of a display-only field. The aggregate function values are computed over the current list.

The aggregate functions are as follows:

COUNT	the number of rows
TOTAL	the arithmetic sum of the values of *tagname*
AVERAGE	the average of the values of *tagname* (AVERAGE can also be written as AVG)
MAX	the maximum value of *tagname*
MIN	the minimum value of *tagname*

- The keyword TODAY that returns today's date
- The keyword CURRENT that returns the current date and time
- Any combination of the preceding functions, combined with the arithmetic operators +, -, *, and /

For more information about aggregate functions, see the *Informix Guide to SQL: Syntax*.

- An expression can contain parentheses to make explicit the precedence of the arithmetic operators.

The following example is from the **sample** form:

```
after add update query of items
   if (total of i19) <= 100 then
      let d1 = 7.50
      else
      let d1 = (total of i19) * .04

   let d2 = (total of i19) + d1
```

After you press ESCAPE following an Add, Update, or Query of the **items** table, PERFORM calculates values for the **d1** (Ship Charge) and **d2** (Total Order Amount) fields. If the value of the **i19** (Total Price) field (all items in the order) is less than or equal to 100, then the value of the **d1** field (Ship Charge) is set to 7.50; otherwise the value is set to the sum of the **i19** (Total Price) field times .04.

The value of the **d2** (Total Order Amount) field is set to the sum of the **i19** (Total Price) field plus the value in the **d1** (Ship Charge) field.

Additional examples of the uses of the LET statement follow:

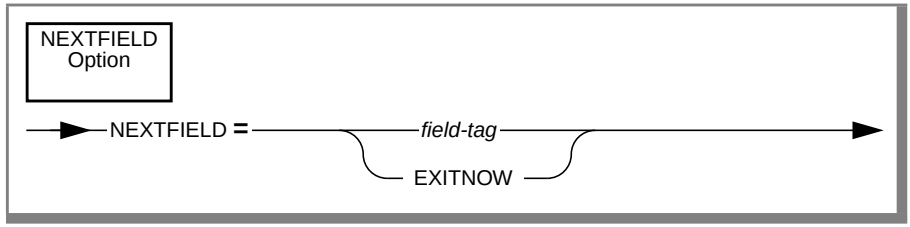
```
let f1 = f2 * 1.065

let s2 = "default string"

let f3 = (f5 + f8) * f7

let ftax = 0.065 * f_price

let f9 = average of f_price

let yr = (today - hdate)/365
```

GLS

The conversion of numeric or monetary values to character strings through the LET statement is influenced by GLS settings. Both the default conversion and the conversion with a USING clause insert locale-specific separator and currency symbols into the created strings, not US English symbols. ♦

NEXTFIELD

When you use the EDITADD or EDITUPDATE options, use the NEXTFIELD action to direct the movement of the cursor. The NEXTFIELD action overrides the default progression as determined by the ATTRIBUTES section of the form specification file.



NEXTFIELD	is the keyword that instructs PERFORM to move the cursor to a particular field or to end the current Add or Update.
field-tag	is the field tag that corresponds to a database column in the active table.
EXITNOW	is a keyword value for NEXTFIELD that ends the current editing.

Usage

- You cannot change the active table by using the NEXTFIELD action to move the cursor to the field of a column in a new table.
- The NEXTFIELD EXITNOW action compares to the ABORT action in the following respects:
  - With NEXTFIELD EXITNOW, PERFORM executes an Update, Remove, or Add, and then exits to the PERFORM menu. NEXTFIELD EXITNOW is equivalent to pressing the ESCAPE key.
  - With ABORT, PERFORM exits to the PERFORM menu without executing an Update, Remove, or Add.
- Because the NEXTFIELD action controls the movement of the cursor, it is effective only after the EDITADD and EDITUPDATE options.

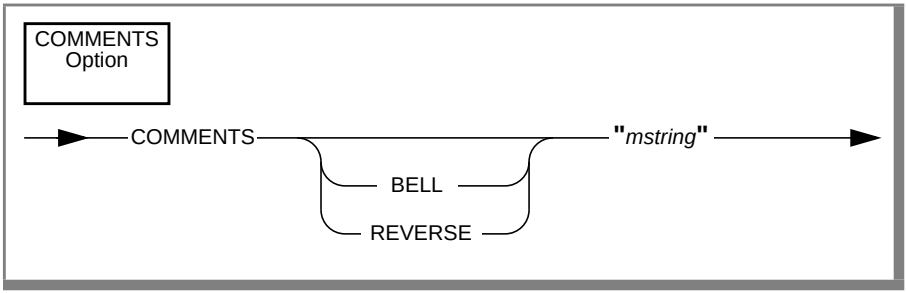
An example from the **sample** form follows:

```
before editadd editupdate of orders
   nextfield = o20
```

In this example, as soon as you indicate that you want either the Add or Update options when **orders** is the active table, PERFORM is instructed to move the cursor to the **o20** column (Customer P.O.). Without this instruction, the cursor would go first to the **o11** field (Order Number) because it is the first **orders** field to appear in the ATTRIBUTES section of the form.

COMMENTS

Use the COMMENTS action to display a message on the Status line of the screen. This use of COMMENTS contrasts with the COMMENTS attribute included in the ATTRIBUTES section that writes a message on the Comment line.



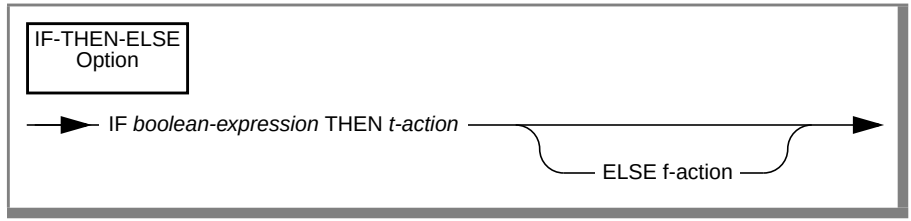
COMMENTS	is the keyword that directs PERFORM to write a message on the Status line.
BELL	is the keyword that directs PERFORM to ring the bell as it writes the message.
REVERSE	is the keyword that directs PERFORM to write the message in reverse video. The default is normal video.
mstring	is the message (string) and must be enclosed in quotation marks. It must fit on one screen line and on one line of the form specification file.

Usage

- If you use the REVERSE keyword, you must take care on some monitors to account for the space required at the beginning of the line for the control characters.
- The message is cleared at the next keystroke. Because PERFORM writes a message whenever a row is written, an Update or an Add is aborted, or a Query or Remove is made, this action is useful only for the EDITADD and EDITUPDATE keywords.

IF-THEN-ELSE

Use the IF-THEN-ELSE action to take actions that depend on the values in the displayed fields.



IF	is a required keyword.
boolean-expression	is a Boolean expression involving field tags that can take on the values true and false.
THEN	is a required keyword.
t-action	is the action or actions to be taken if *boolean-expression* is true.
ELSE	is a keyword.
f-action	is the action or actions to be taken if *boolean-expression* is false.

Usage

- A Boolean expression is a combination of logical comparisons (=, <>, >, <, >=, <=) and logical operations (AND, OR, NOT) among expressions as previously defined. You can also use the operators IS NULL and IS NOT NULL.
- For CHAR type fields only, a Boolean expression may be also of the form

  ```
  field-tag MATCHES "string"
  ```

  where *string* must be enclosed within quotation marks and can include wildcard characters as defined in the *Informix Guide to SQL: Syntax*.
- *t-action* and *f-action* are either single actions as defined in this section or more than one such action between the keywords BEGIN and END.

The following example illustrates a simple IF-THEN-ELSE with one true action and one false action:

```
if (f1 * f2 > 200) then LET f5 = -f4
                   else LET f5  = -5
```

A more complex example from the **sample** form specification file follows:

```
after add update query of items
   if (total of i19) <= 100 then
      let d1 = 7.50
      else
      let d1 = (total of i19) * .04

   let d2 = (total of i19) + d1
```

After you press ESCAPE following an Add, Update, or Query of the **items** table, PERFORM calculates values for the **d1** (Ship Charge) and **d2** (Total Order Amount) fields. If the value of the Total Price field (all items in the order) is less than or equal to 100, then the value of the **d1** field (Ship Charge) is set to 7.50; otherwise, the value is set to the sum of the Total Price field times .04.

The value of the **d2** (Total Order Amount) field is set to the sum of the Total Price field (all items in the order) plus the value in the Ship Charge field.

GLS

The results that character comparisons and OF MATCHES, LIKE and BETWEEN expressions contain character arguments are dependent on GLS collation settings. Refer to Appendix C, "Global Language Support," and the *Informix Guide to GLS Functionality*. ♦

The SAMPLE Form Specification File

The **sample** form specification file was designed for entering and maintaining data on customers and orders listed in the **stores7** demonstration database. The first screen displays information from the **customer** table and is labeled CUSTOMER INFORMATION. The second screen is used to enter and retrieve information about customer orders.

```
database stores7

screen
{
================================================================================
================================================================================
CUSTOMER INFORMATION:

Customer Number: [c1         ]

        Company: [c4                  ]
     First Name: [c2        ]              Last Name: [c3             ]

        Address: [c5                  ]
                 [c6                  ]

           City: [c7             ]  State: [c8]   Zip: [c9   ]
      Telephone: [c10              ]


================================================================================
================================================================================
}

screen

{
================================================================================
CUSTOMER NUMBER: [c1         ]           COMPANY: [c4                  ]

ORDER INFORMATION:
     Order Number: [o11         ]       Order Date: [o12          ]

     Stock Number: [i13        ]      Manufacturer: [i16]
      Description: [s14            ]                [m17            ]
             Unit: [s16            ]
                                            Quantity: [i18        ]
                                           Unitprice: [s15               ]
                                         Total Price: [i19               ]

SHIPPING INFORMATION:
 Customer P.O.: [o20       ]           Ship Charge: [d1        ]

       Backlog: [a]             Total Order Amount: [d2        ]
```

```
     Ship Date: [o21       ]
     Date Paid: [o22       ]
  Instructions: [o23                                    ]

}
end
tables
customer items stock
orders manufact

attributes
c1 = *customer.customer_num
   = orders.customer_num;
c2 = fname,
       comments = "Please enter initial if available";
c3 = lname;
c4 = company, reverse;
c5 = address1;
c6 = address2;
c7 = city;
c8 = state, upshift, autonext,
     include = ("CA","OR","NV","WA"),
     default = "CA" ;
c9 =  zipcode, autonext;
c10 = phone, picture = "###-###-####x####";
o11 = *orders.order_num = items.order_num;
o12 = order_date, default = today, format = "mm/dd/yyyy";
i13 = items.stock_num;
    = *stock.stock_num, noentry, noupdate, queryclear;
i16 = items.manu_code, lookup m17 = manufact.manu_name
       joining *manufact.manu_code, upshift, autonext;
    = *stock.manu_code, noentry, noupdate,
          upshift, autonext, queryclear;
s14 = stock.description, noentry, noupdate;
s16 = stock.unit_descr, noentry, noupdate;
s15 = stock.unit_price, noentry, noupdate;
i18 = items.quantity, include = (1 to 50),
    comments = "Acceptable values are 1 through 50" ;
i19 = items.total_price;
o20 = po_num, required,
    comments = "If no P.O. Number enter name of caller" ;
a   = backlog, autonext;
o21 = ship_date, default = today, format = "mm/dd/yyyy";
o22 = paid_date, format = "mm/dd/yyyy";
o23 = ship_instruct;
d1 =  displayonly type money;
d2 =  displayonly type money;

instructions

customer master of orders;
orders master of items;
composites <items.stock_num, items.manu_code>
            *<stock.stock_num, stock.manu_code>

before editadd editupdate of orders
   nextfield = o20

before editadd editupdate of items
   nextfield = i13
```

```
after editadd editupdate of quantity
   let i19 = i18 * s15
   nextfield = o11

after add update query of items
   if (total of i19) <= 100 then
      let d1 = 7.50
      else
      let d1 = (total of i19) * .04

   let d2 = (total of i19) + d1

after display of orders

   let d1 = 0
   let d2 = 0

end
```

The CUSTOMER INFORMATION Screen

The CUSTOMER INFORMATION screen contains fields for the entry and display of all the columns in the **customer** table. You can use this screen to add or remove a customer from the database.

This screen has the following important points:

- The **customer** table is joined with the **orders** table at the **customer_num** column. The **customer** table is the dominant table in this join.
- A Comment line appears when the cursor moves into the **c2** (First Name) field.
- The **c4** (Company) field appears in reverse video.
- PERFORM automatically enters uppercase letters into the **c8** (State) field. The default value for the field is `CA`, and only abbreviations for four states are allowed.
- A character pattern is specified for the **c10** (Telephone) field.

The ORDER INFORMATION Screen

The second screen contains fields drawn from the **orders**, **items**, **stock**, and **manufact** tables. This screen is used to enter information about a customer's order. Shipping information (purchase order number, instructions, date sent, and so on), and order information (order number, date, items included in the order, total prices on each item, and so on) are entered on this screen.

This screen has the following important points:

- The **c1** (Customer Number) and **c4** (Company) fields are repeated from the CUSTOMER INFORMATION screen.

  The verify join between the **customer.customer_num** column and the **orders.customer_num** column prevents the assignment of an order to a nonexisting customer. When the **orders** table is active, no value can be entered into a field that does not already exist in the **customer** table.
- The **orders** and **items** tables are joined at the **order_num** column. This verify join prevents the assignment of an item to a nonexisting order number. When the **items** table is active, no value can be entered into the field that does not already exist in the **orders** table.
- The **o12** (Order Date) field has an assigned format and defaults to the current date.
- The **stock_num** column in the **items** table is joined with the **stock_num** column in the **stock** table. This is a verify join.
- The **manu_code** column in the **items** table is joined with the **manu_code** column in the **stock** table. This is a verify join.

- The **i13** (Stock Number) and **i16** (Manufacturer) fields are members of a composite join between the **items** and **stock** tables. This is a composite verify join, so no values can be entered in the **stock_num** and **manu_code** fields (when the **items** table is active) that do not already exist in those two columns in the **stock** table. This precludes entry of stock numbers and manufacturer codes that individually exist in the database but, as a composite, do not correspond to a unique row in the **stock** table.

  To specify a unique row in the **stock** table requires both the **stock_num** and **manu_code**. For example, the **stock** table contains three rows with the stock number 1 and four rows with the manufacturer code HRO. Knowing the stock number or manufacturer code alone does not allow you to locate a unique row. You need both the stock number (1) and the manufacturer code (HRO) to specify a unique row (baseball gloves produced by Hero) in the table.
- Once the **i13** (Stock Number) and **i16** (Manufacturer) fields are filled, PERFORM can locate the corresponding unique row in the **stock** table. The **s14** (Description), **s16** (Unit), and **s15** (Unitprice) fields automatically display this information.
- The **i16** (Manufacturer) field is involved in a lookup join that locates the appropriate manufacturer name in the **manufact** table and places this information in the **m17** field.
- The **i18** (Quantity) field allows the entry of values 1 through 50 only, and it displays a comment when the cursor moves into the field. This helps to prevent mistaken entries of an extra digit (for example, 100 in place of 10).
- The cursor does not visit the **o11** (Order Number) field when the **orders** table is the active table because the **order_num** column in the **orders** table is a SERIAL data type.
- The following entry in the INSTRUCTIONS section tells PERFORM that when the **orders** table is the active table, the cursor first goes to the **o20** (Customer P.O.) field, rather than the **o12** (Order Date) field:

  ```
  before editadd editupdate of orders
     nextfield = o20
  ```
- The default value for field **o21** (Ship Date) is set to today.

- The **d1** and **d2** fields do not correspond to any database columns. One of these is the Ship Charge field and is the total shipping charge for the entire order. The second is the Total Order Amount field and is the total charge for the entire order, including all items and the shipping charge. PERFORM calculates each field automatically. The following entry in the INSTRUCTIONS section tells PERFORM that if the total of all values in the field **i19** for this order is less than or equal to 100, then set the value in the field **d1** to 7.50. If the total of all values in the field **i19** for this order is greater than 100, then set the value in field **d1** to the product of this total times .04.

  ```
  after add update query of items
     if (total of i19) <= 100 then
        let d1 = 7.50
        else
        let d1 = (total of i19) * .04

     let d2 = (total of i19) + d1
  ```

  The Total Order Amount (the **d2** field) is the sum of all values in Total Price (the **i19** field) for the order plus the Shipping Charge (the **d1** field).

  The total price of an individual item in a customer order is calculated automatically by FORMBUILD. This field is filled in by PERFORM as soon as you supply information for the fields **i13** (the Stock Number of the item), **i16** (the Manufacturer of the item), and **i18** (the Quantity of the item). PERFORM can do this because, in the INSTRUCTIONS section, PERFORM is told to calculate the value of the Total Price (the **i19** field) based on the values entered into the **i18** (the Quantity) and **s15** (the Unit Price) fields.

- The **customer** table is the master of the **orders** table. You can easily query the **orders** table (and locate all orders placed by a customer) based on the current row in the **customer** table by selecting the Detail option.
- The **orders** table is the master of the **items** table. You can easily query the **items** table (and locate all items contained in each order) based on the current row in the **orders** table by selecting the Detail option.

Chapter

3

The PERFORM Screen Transaction Processor

In This Chapter

PERFORM is an INFORMIX-SQL program designed to streamline data entry and retrieval. After you create a screen form with FORMBUILD, you can use the form with PERFORM to query and modify the data in a database. For information about designing and building screen forms, see Chapter 2, "The FORMBUILD Transaction Form Generator."

This chapter is divided into two parts. The first part describes the following PERFORM procedures:

- Accessing PERFORM from the Main menu
- Running operating-system commands while using PERFORM
- Using special keys to position the cursor and edit text
- Entering and editing data on the screen
- Data checking with PERFORM
- Controlling user privileges in PERFORM
- Using the current list

The second part of this chapter discusses each PERFORM option. The options are listed in alphabetical order.

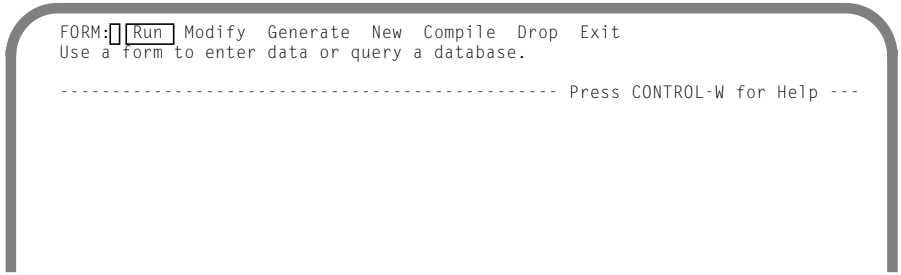
Running PERFORM

PERFORM uses the file that FORMBUILD generates when you compile a form specification file. This file must be in your working directory or a directory included in your DBPATH environment variable.

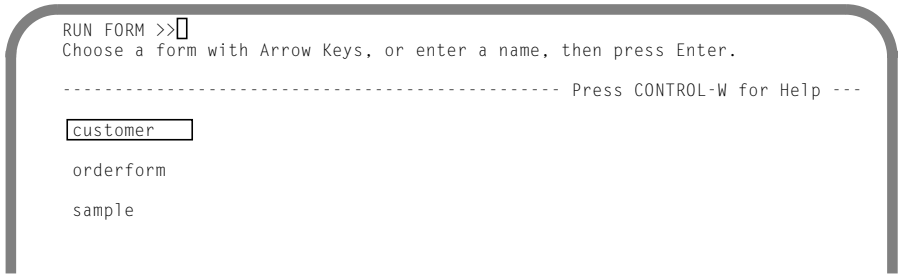
You can use PERFORM from the INFORMIX-SQL Main menu or directly from the operating system. For information about command-line usage, see Appendix G, "Accessing Programs from the Operating System."

Accessing PERFORM from the Main Menu

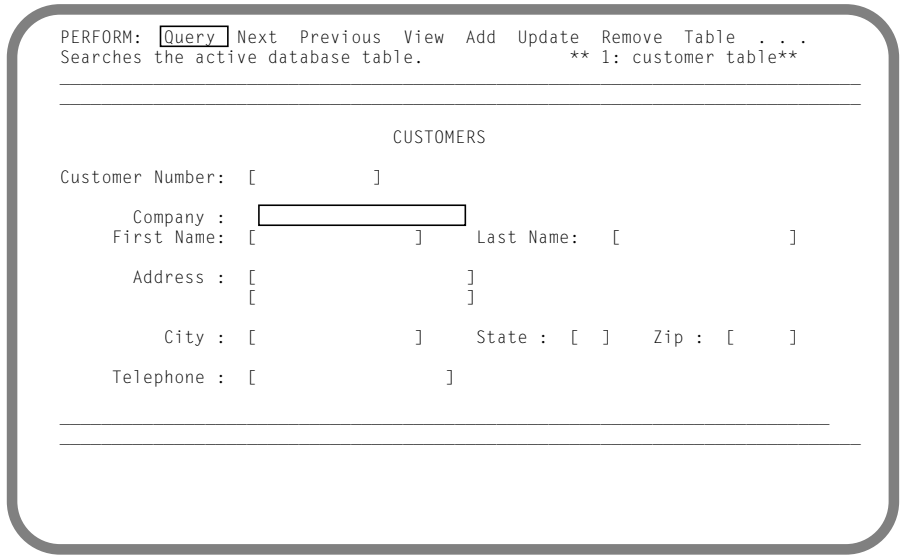
Select the Form option on the INFORMIX-SQL Main menu. The FORM menu is displayed.



Select the Run option on the FORM menu. The RUN FORM screen is displayed with a list of available screen forms.



Type the name of the form you want to use or use the Arrow keys to highlight your choice on the screen. Press RETURN. The form you select appears on the screen with the PERFORM menu, as shown in the following figure.



The PERFORM Screen

The PERFORM screen is divided into three sections:

- The first two lines of the screen (the Information lines) display the PERFORM menu options, a message describing the highlighted option, and the number and name or alias of the active table.
- The middle section of the screen (the screen form) displays the form you selected.
- The bottom two lines of the screen (the Comment line and Status line) display PERFORM messages, as well as comments specified in the form file.

The Information Lines

The PERFORM menu is two pages long. The first Information line displays a list of menu options; the second Information line describes the current option and indicates the number and name or alias of the active database table. The next two screens illustrate the Information lines on the two-page PERFORM menu.

```
PERFORM:  Query [Next] Previous  View  Add  Update  Remove  Table  Screen  . .
 .
Searches the active database table.              ** 1: customer table**
```

```
PERFORM: . . . [Current]  Master  Detail  Output  Exit
Displays the current row of the current table.          ** 1: customer table**
```

The ellipsis on the first menu page indicates that additional menu items are available on the second menu page. The ellipsis on the second menu page indicates that additional menu items are available on the previous menu page.

Tip: *The number of options that appears on the first menu page depends on the character capacity of your screen. The two-page screens displayed here demonstrate a terminal or monitor with an 80-character screen. Terminals with a larger character capacity show more options on the first menu page.*

Use the SPACEBAR or the Arrow keys to move the highlight onto the menu options. When you move the highlight past the first or last menu option on a page, the alternate menu page appears; the menu does not scroll. The highlight never rests on the ellipses; when you move the highlight past the last or first option on each screen page, the next PERFORM menu page appears.

PERFORM is a menu-driven program. To work with the data on the screen, select one of the menu options. You select a menu option on the first Information line by using the Arrow keys or the SPACEBAR to position the highlight on a menu option and then pressing RETURN, or by typing the first letter of the menu option. PERFORM immediately displays the screen for the selected option. If you want to return to the menu without making any entries, press the Interrupt key. This key is DEL on most systems.

The PERFORM screen has the following menu options:

Query	retrieves rows from the database based on search values you enter on the form and stores the rows in the current list.
Next	displays the next row in the current list.
Previous	displays the previous row in the current list.
View	displays the contents of a field of data type TEXT or BYTE.
Add	adds data to the database.
Update	modifies data in the database.
Remove	deletes a row from a database table.
Table	displays a different table in the form.

Screen displays a different screen page of the form.

Current restores the base current list in multitable queries and displays the most up-to-date version of the displayed row in multiuser environments.

Master displays the master table of the active table.

Detail displays the detail table of the active table.

Output writes the selected row or rows to an operating system file in either Screen or Unload format.

Exit leaves PERFORM.

The Information lines also indicate the number and name or alias of the active table. Every table included in the screen form has a table number assigned according to the order in which display field tags (including joins) for the table first appear in the ATTRIBUTES section of the form specification file. This number appears next to the table name in the right-hand corner of the second Information line when the table is active. The table number is useful for nonsequential moves to another table using the Detail and Table options.

The Screen Form

The screen form consists of one or more display fields in which PERFORM displays—and you enter—data. Each display field on a screen form corresponds to one or more of the database columns or to a display-only field specified in the form file. Unless you specify alternative delimiters in the form specification file, active display fields are surrounded on the screen by brackets ([ ]). Fields with no delimiters are not active; values may appear in them if they are LOOKUP fields or display-only fields, but you cannot enter data into them.

A screen form may be one page or several pages long and can contain columns from several tables. All tables included in a form must be part of the same database.

Here is how the PERFORM screen looks when you use the **customer** form included with the demonstration database.

```
PERFORM: Query  Next  Previous  View  Add  Update  Remove  Table  . . .
Searches the active database table.                ** 1: customer table**
_____________________________________________________________________________
_____________________________________________________________________________

                                 CUSTOMERS

Customer Number:  [           ]

       Company :  [                    ]
     First Name:  [               ]     Last Name:   [               ]

       Address :  [                   ]
                  [                   ]

          City :  [               ]     State :  [  ]    Zip :  [     ]

     Telephone :  [                  ]

_____________________________________________________________________________
_____________________________________________________________________________
```

Status Lines

PERFORM uses the last two lines of the screen to display PERFORM error messages, as well as any messages generated by the form itself.

```
The two entries were not the same - please try again.
```

Running Operating-System Commands from PERFORM

To run operating-system commands from the PERFORM menu, press the exclamation point (!) key. PERFORM displays the exclamation point at the bottom of the screen. Enter an operating-system command and press RETURN. PERFORM displays the results of your command and then the following message:

```
Press RETURN to continue
```

Press RETURN to leave the operating system and return to PERFORM.

Entering Data

Use the Add and Update options to enter data directly into the database from the screen form. You must enter data of the type specified when the table was created—dates in DATE fields, money in MONEY fields, and so on. If you make a mistake entering data, you can use the field-editing keys to correct it. (See "Field Editing" on page 3-14.)

Data Types

The following list discusses the kind of data to enter for each data type. If you enter data of the wrong type, PERFORM displays the following message on the Status line:

```
Error in field
```

Enter an acceptable value or press the Interrupt key to cancel the option you are using. You can use the Info options on the SQL or TABLE menu to find out the data type for each column in a table. For more information about data types, see the *INFORMIX-SQL User Guide*.

BYTE	You can use the View option of the PERFORM menu to display BYTE fields that are referenced in your form with the PROGRAM attribute. You can display but not change the contents of a BYTE field.
CHAR[(*n*)]	Enter letters, numbers, and symbols. During an Add or Update, the character data string can be as long as the display field.
CHARACTER	is a synonym for CHAR.
SMALLINT	Enter a whole number from -32,767 to +32,767.
INTEGER	Enter a whole number from -2,147,483,647 to +2,147,483,647.
INT	is a synonym for INTEGER.
SERIAL	PERFORM assigns SERIAL values automatically, so you never add data to a SERIAL field or update it. However, you can enter search values in SERIAL fields when you use the Query option.
DECIMAL[(*m*,[*n*])]	Enter decimal numbers. The format of the number (number of places to the right and left of the decimal point) depends on the format you specified when you created the database table. If you enter a number with too many spaces after the decimal point, PERFORM rounds it off.
DEC	is a synonym for DECIMAL.
NUMERIC	is a synonym for DECIMAL.
MONEY	Enter dollars-and-cents amounts without dollar signs and commas (for example, `4254.30` not `$4,254.30`). When you press RETURN, PERFORM automatically adds a dollar sign.

SMALLFLOAT	Enter floating point numbers with up to 7 significant digits. PERFORM sometimes introduces a slight discrepancy when you type numbers into SMALLFLOAT fields. The entry `1.1`, for example, might display as `1.11000001` after you press RETURN or ESCAPE. This occurs because of the way a computer stores numbers internally, and only affects you when a precision of more than 7 digits is required.
REAL	is a synonym for SMALLFLOAT.
FLOAT[(*n*)]	Enter floating-point numbers with up to 14 significant digits. The discrepancies mentioned for SMALLFLOAT data also apply to this data type if you require a precision of more than 14 digits.
DATE	Enter dates in the form `[mm]m[d]d[yy]yy`, with any nonnumeric characters as optional dividers (for May 2, 1985, you could enter `May 2 85`, `08/02/85`, `8.2.85`, or `08 02 1985`).
DATETIME	Enter DATETIME values in the form `[yyy]y-[m]m-[d]d [h]h:[m]m:[s]s.[ffff]`. You must separate the fields as follows: YEAR-MONTH, MONTH-DAY, DAY(space)HOUR, HOUR:MINUTE, MINUTE:SECOND, SECOND.FRACTION.
INTERVAL	Enter INTERVAL values in the form `[yyy]y-[m]m` or `[d]d [h]h:[m]m:[s]s.[ffff]f`. You must separate the fields as follows: YEAR-MONTH, DAY(space)HOUR, HOUR:MINUTE, MINUTE:SECOND, SECOND.FRACTION.
TEXT	You can only display the contents of a TEXT field; you cannot change it.
VARCHAR	Enter letters, numbers, and symbols. During an Add or Update, the character data string can be as long as the display field.

GLS

The numeric and decimal separators can be tailored using GLS settings. These settings change the separators displayed on the screen in a numeric or monetary field. For example, *1234.56* will display as *1234,56* in a French or German locale. Also, in the French or German locale values input by the user will be expected to contain commas, not periods, as decimal separators.

The installation of message files in a subdirectory of **$INFORMIXDIR/msg** and subsequent reference to that subdirectory by way of the environment variable **DBLANG** causes DATETIME and DATE values to display locale-specific month name abbreviations on the form. Similarly, month name values are expected to be valid locale specific names when input. For example, the month name June in a French locale would have to be input as the month name abbreviation *Jui*, which stands for *Juin* (the French word for June), rather than *Jun*. If you are unsure about the correct month name, specify months numerically. ♦

Special Functions

As you enter data or a query, three special functions are available by using selected keys.

Function	Key Used
Help	CONTROL-W displays a HELP screen that contains a short summary of special keys, control keys, editing keys, and other information about PERFORM.
Execute	ESCAPE runs the option you select. To add a new row, type `a` to select the Add option, enter the information for the row, and press ESCAPE to add the row to the database.
Interrupt	On most systems DELETE or CONTROL-C interrupts or cancels the option you are using. For example, if you select Add when you really want Query, press CONTROL-C and then select the Query option.

Positioning the Cursor

You can use the following keys to position the cursor on the screen.

Movement	Key Used
Next Field	The RETURN and [↓] keys move the cursor to the next field.
Backspace	The BACKSPACE and [←] keys move the cursor backward one character at a time without erasing any text. Pressing either key at the beginning of a field moves the cursor to the previous field.
Forward	The [→] key moves the cursor forward one character at a time without erasing any text. Pressing the [→] key at the end of a field moves the cursor to the next field.
Fast Forward	CONTROL-F moves the cursor down the screen rapidly, stopping in the first field on each line. Use CONTROL-F to move quickly to the bottom of a form that contains many fields.
Fast Backspace	CONTROL-B moves the cursor up the screen rapidly, stopping at the last field on each line. Use CONTROL-B to move quickly to the top of a long form.

Field Editing

If you make a mistake entering data in a field, you can correct it by backspacing and retyping. However, you might find it faster to use the PERFORM field-editing feature. You can use two editing modes to enter data into a field:

- In *typeover mode*, the characters you type replace existing data. For example, you could use typeover mode to change "Sports 'R Us" to "Abe's Sporting Goods."
- In *insert mode*, the characters you type push existing data to the right. For example, you could use insert mode to add an *i* to *Rchard*.

Whenever the cursor enters a field, PERFORM is in typeover mode; you must press the Insert key to activate the insert mode. Press the Insert key or CONTROL-A a second time to return to typeover mode. Move the cursor into a new field, and you are automatically in typeover mode.

Use the following keys to edit data that appears in a field.

Function	Key Used
Backspace	The BACKSPACE and [←] keys move the cursor back one character at a time without erasing any text. If you press either key at the beginning of a field, the cursor moves back to the previous field.
Forward	The [→] key moves the cursor forward one character at a time without erasing any text. If you press [→] at the end of a field, the cursor moves forward to the next field.
Delete a Character	Delete or CONTROL-X deletes the character beneath the cursor. The cursor remains in place, and text shifts over to fill the space that was occupied by the deleted character.
Change Mode	Insert or CONTROL-A shifts between insert and typeover mode. When you access PERFORM, you are in typeover mode.
Delete Forward	CONTROL-D deletes everything from the current cursor position to the end of the field.
Repeat Data	CONTROL-P enters the most recently displayed value in a field. When you use the Add option to enter several rows in which one or more fields contain the same data, you can avoid retyping the data by pressing CONTROL-P. When you use the Update option, CONTROL-P restores the value that appeared in a field before you modified the field.
Clear Screen	CONTROL-C clears any search criteria you have entered with the Query option.

Using the Multiline Editor

A multiline editor is available for editing long character fields, also called multiline fields. A multiline field has more than one physical field, as shown in the following form specification file.

```
database reference

screen
{
    TITLE: [b001                                          ]
   AUTHOR: [b002                                ]
 SYNOPSIS: [b003                                          ]
           [b003                                          ]
           [b003                                          ]
           [b003                                          ]
}

tables
booktab

attributes
b001 = refdpt.booktab.title
b002 = refdpt.booktab.author
b003 = refdpt.booktab.synopsis,WORDWRAP COMPRESS
  .
  .
  .
```

You invoke the multiline editor by using the WORDWRAP attribute (see "WORDWRAP" on page 2-63 for detailed information). Most keys function the same in multiline editing as they do in normal field editing, with a few exceptions.

RETURN	RETURN causes the cursor to leave the current multiline field and move to the first position in the next field.
Up Arrow	The [↑] key moves the cursor one line up within the same multiline field. The cursor moves to the left if necessary to avoid editor blanks (see "WORDWRAP" on page 2-63). If the cursor is on the top line of a multiline field, the [↑] key moves the cursor to the first character position in the preceding field.
Down Arrow	The [↓] key moves the cursor one line down within the same multiline field. The cursor moves to the left if necessary to avoid editor blanks. If the cursor is on the bottom line of a multiline field, the [↓] key moves the cursor to the first character position in the following field.
TAB	If you are in typeover mode, TAB moves the cursor to the next field.
CONTROL-N	CONTROL-N inserts a newline character, causing subsequent text to move to the first position in the following line of the same multiline field. This could cause text to ripple down toward the bottom of the field, and you might lose the text that was in the last line of the field.

Display Field Order

The cursor ordinarily moves through the display fields in the order in which their field tags are listed in the ATTRIBUTES section of the form file. You can modify this order by using a NEXTFIELD statement in the INSTRUCTIONS section of the form file.

Data Checking

The attributes and instructions in the form file can affect data entry, data storage, data display, and cursor movement when you use the Add, Update, and Query options. If you get undesired displays or cursor movement, you can modify the form file. The effects of some attributes and instructions are listed here, followed by the relevant options. For details about attributes and instructions, see Chapter 2.

- The case of the character data on the screen is different from what you type. Check for UPSHIFT and DOWNSHIFT attributes (Add, Update, Query).
- SMALLFLOAT or FLOAT data on the screen is different from what you type. Check for a FORMAT attribute that causes rounding off by specifying the number of places to the right and left of the decimal point (Add, Update).
- PERFORM displays the following message:

  ```
  This value is not among the valid possibilities.
  ```

  Check for an INCLUDE attribute that specifies acceptable values and ranges of values (Add, Update).
- The terminal beeps and does not echo your entry on the screen. Check for a PICTURE attribute that limits data entry to a specified pattern of variables and literals (Add, Update).
- The cursor skips over a bracketed display field. If the field is not a SERIAL field, check for a NOENTRY attribute (for Add only), a NOUPDATE attribute (for Update only), or a NEXTFIELD action (both Add and Update).
- PERFORM displays the following message:

  ```
  This field requires an entered value.
  ```

  Check for a REQUIRED attribute (Add). You must explicitly enter all values for a field with the REQUIRED attribute unless you have specified a value with DEFAULT.
- PERFORM displays the following message:

  ```
  Please type again for verification.
  ```

  Check for a VERIFY attribute (Add, Update).

- Data you enter appears on the screen justified to the right. Check for a RIGHT attribute (Add).
- Number data appears on the screen justified to the right and padded with leading zeros. Check for a ZEROFILL attribute (Add).
- A value you did not enter appears in a field. Check for a DEFAULT attribute, a PICTURE attribute with literals (Add, Query), a joined field (Add, sometimes Query), or a LET action (Add).
- The cursor moves automatically to the next field after this field is full. Check for an AUTONEXT attribute (Add, Update).
- A line of text appears on the screen. Check for a COMMENTS attribute (Add, Update, Query).
- A line of text appears on the screen, in regular or reverse video, and / or the terminal bell rings. Check for a COMMENTS action (Add, Update, Query).
- Data stores automatically before you press ESCAPE. Check for a NEXTFIELD EXITNOW action (Add, Update).
- PERFORM displays the following message:

  ```
  This is an invalid value--it does not exist in
  tablename.
  ```

  Check for a verify join (Add, Update).

User Access Privileges

Access privileges controlled by the GRANT and REVOKE statements can affect your ability to display, enter, modify, and remove data for a table or a display field. A message like the following means that you do not have access privileges:

```
Permission not granted to allow update
```

Use the Info option on the SQL or TABLE menu to find out the access privileges for a particular table.

See the *Informix Guide to SQL: Syntax* for more information about privileges.

The Current List

The current list is a temporary storage area where PERFORM stores the results of a query. It can hold from one row to all the rows in a database. Whenever you select the Query option, PERFORM erases the existing current list to make room for new query results.

The Query, Next, Previous, Remove, and Update options all involve the current list. The Query option finds all rows that satisfy the search conditions and puts them in the current list. The Next and Previous options step through the rows in the current list in sequential order. The Update and Remove options can only work with rows in the current list.

Menu Options

The menu options you can use with PERFORM are described in detail on the following pages. They are listed in alphabetical order, rather than menu order, for easy reference.

ADD

Use the Add option to create new rows in the active table. You can type data on a screen form, review it, edit it, and store it in the database.

Use the following procedure for the ADD option:

1. Type `a` to select the Add option.

   PERFORM clears all data from the screen form except joined fields and some display-only fields, displays spaces or default values in the fields, and positions the cursor in the first field.

2. Fill in the form with the values you want to enter.

   If you enter data inappropriate to the data type of the display field, PERFORM displays the following message:

   ```
   Error in field
   ```

   You cannot move on to the next display field until you correct the entry.

3. Press ESCAPE to store the row or the program Interrupt key to cancel the addition and redisplay the PERFORM menu options.

Usage

- If PERFORM displays this message, it does not store a row when you press ESCAPE:

  ```
  Could not insert new row - duplicate value in a unique
  index column
  ```

  You are trying to enter a duplicate value where it is not permitted. Use the Info option on the SQL or TABLE menu (or execute an INFO statement) to check for unique indexes.

- Tables may be fully or partly unavailable to you because another user has invoked the LOCK TABLE statement or because another user is updating a row that you attempt to update or remove. In such a case, PERFORM displays an error message.

- PERFORM sometimes introduces a slight discrepancy when you enter numbers in SMALLFLOAT or FLOAT fields. The entry `1.11`, for example, might display as `1.11000001` after you press RETURN or ESCAPE. This discrepancy occurs because of the way a computer stores numbers internally.

Related Options

Update, Remove

CURRENT

The Current option rereads and redisplays the current row in the current list for the active table.

Use the following procedure for the Current option:

1. Type c to select the Current option.
2. PERFORM displays the most up-to-date version of the screen you were looking at before you moved to another table.

Usage

The Current option is useful in two situations:

- In a LAN environment, another user can modify the information corresponding to a display field on your screen. When you use the Current option, PERFORM rereads the row, displaying the most recent information.
- When a form includes a join field, each table represented on the screen form has its own current list. Looking at the information in the active table might make you "lose your place" in one or more of the other current lists. The Current option returns you to your original position in the current list of the active table.

DETAIL

The INSTRUCTIONS section of the form file can include one or more master-detail table relationships for tables with join fields to simplify multitable queries. The Detail option automatically selects, displays, and queries the detail table of the active table.

Use the following procedure for the Detail option:

1. Enter `d` to select the Detail option.
2. If the active table has no detail table, PERFORM displays an error message. If the active table has one or more detail tables specified, PERFORM locates and displays the first detail table of the active table, no matter what the absolute table number of the detail table is.
3. PERFORM automatically runs a query on the detail table, using the current values in the fields in the master table that join the fields in the detail table as search values. PERFORM then puts the rows that satisfy the query conditions in the current list of the detail table. You can use the Next and Previous options to examine the rows. If no rows are found, PERFORM displays the following message:

   ```
   There are no rows satisfying the conditions.
   ```

4. Type `m` to make the master table the active table again. You will see the first master-table row with join-field values that match the current values in the detail table.

Usage

- If no explicit master / detail relationship exists, PERFORM displays an error message when you use the Master option or the Detail option without a table number.
- If more than one detail table has been specified for a master table in the INSTRUCTIONS section, type `d` to display and query the first detail table; type `d` preceded by the number of another detail table to display and query the other detail tables.
- You use a table number to query any *detail* table that joins the active table, even if no master-detail relationship is specified. Type `4d`; if table number 4 joins the active table, PERFORM queries table number 4 and it becomes the new active table. However, PERFORM displays an error message when you type `d` without a table number if no master-detail relationship has been specified in the INSTRUCTIONS section.

Related Option

Master

EXIT

Use the Exit option to exit from PERFORM.

Use the following procedure for the Exit option:

1. Type e to run the Exit option.
2. PERFORM returns you to your starting place, either the FORM menu or the operating system.

MASTER

Use the Master option to move directly from a detail table to its master table.

Use the following procedure for the Master option:

1. Type `m` to select the Master option.
2. If the active table has no master table, PERFORM displays an error message. If the active table has a master table, PERFORM displays the master table with the row found joining the detail table.
3. You must declare a Master-Detail relationship in the INSTRUCTIONS section to use this option.

Related Option

Detail

NEXT

Use the Next option to step forward through the rows in the current list.

Use the following procedure for the Next option:

1. Use the Query option to put the rows you want to inspect in the current list.
2. Type `n` to run the Next option. PERFORM displays the next sequential row in the current list.
3. Type `n` repeatedly. When you reach the last row in the current list, PERFORM displays the following message:

```
There are no more rows in the direction you are going.
```

Usage

If you want to move forward several rows at once, enter a number before the Next option; for example, entering `10n` skips ahead 10 rows.

Related Options

Query, Previous

OUTPUT

You can use the Output option on the PERFORM menu to write one or all rows in the current list to a new or existing file.

You can produce an output file in which rows appear just as they do on your screen, including data, display field labels, boxes, lines, and so on. Alternatively, you can produce an output file in which rows appear just as they do when you run an UNLOAD statement. Rows retrieved using this alternative method appear in an ASCII file, one row per record, with fields separated by the default delimiter. You can use a file in this Unload format with the ACE READ statement to produce a report.

Use the following procedure for the Output option:

1. Select the Query option of the PERFORM menu to retrieve a list of the row or rows that you want to write to a file. If necessary, use the Next or Previous options to display the single row that you want to write to the file.
2. Type `o` to select the Output option. PERFORM prompts you for a filename:

   ```
   Enter output file (default is perform.out):
   ```

3. Press RETURN to accept the default filename. Alternatively, type the name of a file in which to store your output and press RETURN. If you want to store your output in a different directory, make sure you include the complete pathname. The name you enter becomes the new default filename of the Output option for the rest of the session, or until you enter another, different, filename.

   PERFORM displays the FORM OUTPUT FILE menu as follows.

```
FORM OUTPUT FILE: [ Append ] Create
Adds new data to an existing output file
```

4. Type a or press RETURN to append the information to the file that you specified in step 3. Type c to create a new file that contains this information.

Tip: *If you enter an existing filename in step 3 and select the Create option, PERFORM overwrites the old version of the file when you run the Output option. You lose any data stored in the old file.*

PERFORM displays the FORM OUTPUT FILE LIST menu as follows.

```
FORM OUTPUT FILE LIST: [ Current-list ] One-page
Writes the Current List to the file
```

5. Type c or press RETURN if you want to store every row in the current list. Type o to store only the row that currently appears on your screen.

PERFORM prompts you for the format of the output file.

```
OUTPUT FORMAT: [ Unload-format ] Screen-format
Writes the selected output in ascii format
```

6. Press RETURN or type u to store the retrieved row or rows as an ASCII file (Unload-format). Select this option if you plan to use this file in an ACE report or as input for another application.

   Type s if you want to store the retrieved row or rows in a file formatted to look the same as your screen display (Screen-format). The file includes the data and any additional field labels, boxes, lines, or other screen items.

   PERFORM writes the rows to the file. A counter at the bottom of the screen increments as each output row is written to the file:

   ```
   Output record number 1
   ```

   If you select the Current-list option in step 5, PERFORM displays each row that it writes and updates the counter as it does so.

Usage

- If you select the Screen-format option on the OUTPUT FORMAT menu, PERFORM copies one page of a screen form for each row in the current list. To copy a row that occupies more than one screen, you must use the Output option separately with each screen.

  If you want to copy all the screens of a three-screen form, for example, perform the following operations:

  1. Type O, A, C, and S to select the Output, Append, Current-List, and Screen-format options to copy all the first screens in the current list to a file.
  2. Select the Screen option to display the second screen.
  3. Repeat the same Output options to copy the second screen.
  4. Use Screen to display the third screen, and then type O, A, C, and S to append the third screen to the file.

  If the query retrieves multiple rows, the file contains all the first screens, followed by all the second screens, and so on.

- The Unload-format option on the OUTPUT FORMAT menu copies entire rows in unload format, regardless of the number of screens in the form.
- The Unload-format option on the OUTPUT FORMAT menu copies the value of every field listed in the ATTRIBUTES section of the form specification file for each row. Fields appear in an output row in the same order in which the corresponding fields are listed in the ATTRIBUTES section of the form specification file. Look-up fields are appended to the end of the row.

PREVIOUS

Use the Previous option to display prior rows in the current list.

Use the following procedure for the Previous option:

1. Use the Query option to put the rows you want to look at in the current list.
2. Type `n` to display the next row.
3. Type `p` to use the Previous option. PERFORM displays the previous row (in this case, the first row) in the current list.
4. When you reach the first row in the current list, type `p`. PERFORM displays the following message:

   ```
   There are no more rows in the direction you are going.
   ```

5. You can use the Previous option whenever you want to display prior rows in the current list.

Usage

If you want to move backward several rows at once, enter a number before the Previous option; for example, entering `10p` skips back 10 rows.

Related Options

Query, Next

QUERY

Use the Query option to search for database rows and columns with specified values based on search values you enter directly into the display fields on a screen form. You can specify the search criteria with 11 different query operators, including 6 relational operators, 2 range operators, 2 wildcard operators, and highest/lowest value operators. PERFORM finds all the database rows that satisfy the conditions and puts them in the current list. You can use the Next and Previous options to view them.

Use the following procedure for the Query option:

1. Type `q` to select the Query option. PERFORM clears the fields in the active table of all data (except data in joined fields with no QUERY-CLEAR attribute) and puts the cursor in the first field.
2. Enter search values in one or more display fields using the syntax described on page 3-35. To find all the rows in the table, do not enter any search values.

   If a display field is too short to hold the search value you enter, PERFORM creates a workspace at the bottom of the screen. When you press RETURN, PERFORM removes the workspace. The display field contains what you entered in the workspace even though you can only see the part of it that fits in the field.
3. Press ESCAPE to run the query; press the Interrupt key to cancel the query and display the PERFORM menu again. When you run the query, PERFORM searches the active table, puts all the rows that satisfy the conditions in the current list, and displays the first matching row on the screen. A Status line message reads as follows:

   ```
   # row(s) found
   ```

   where # represents the number of rows that contain the specified search value(s) in the specified display field(s). You can use the Next and Previous options to look at the rows in the current list. If no rows satisfy the conditions, the Status line message reads as follows:

   ```
   There are no rows satisfying the conditions
   ```

The following symbols can be used in the specification of queries.

Symbol	Name	Data Types	Pattern
=	equal to	all	=x
>	greater than	all	>x
<	less than	all	<x
>=	greater than or equal to	all	>=x
<=	less than or equal to	all	<=x
<>	not equal to	all	<>x
:	range	all	x:y
..	range	DATETIME	x..y
		INTERVAL	
*	wildcard	CHAR	*x, x*, *x*
?	single-character wildcard	CHAR	?x, x?, ?x?, x??
|	or	all	a | b
>>	highest value	all	>>
<<	lowest value	all	<<

=	The default query operator; if you do not enter another operator, PERFORM assumes the equal sign. Enter the equal sign by itself to search for a database row that contains a null CHAR column; enter =* to find a row that contains a column with an asterisk.
x	Any search value with the appropriate data type for the search field. Enter the search value immediately after any one of the first six query operators in the previous table. Do not leave a space between the query operator and the search value.
>	For CHAR data, *greater than* means later in ASCII order (a>A>1). For DATE or DATETIME data, *greater than* means after. (See Appendix E, "The ASCII Character Set," in this manual for more information.)
<	For CHAR data, *less than* means earlier in ASCII order (1<A<a). For DATE or DATETIME data, *less than* means before.
y	Any search value with a value higher than x.
|	The search operator that signifies *or*. `110|118|112` in the Customer Number field, for instance, would search for rows with the value 110, 118, or 112 in the **customer_num** column.
:	The search operator that specifies a range. You must give the lower value before the search operator and the higher value after the operator in a range query. Queries with the range operator are inclusive. The search criterion `1:10` would find all rows with a value in that column from 1 through 10, inclusive. When an odd number of colons appear in a query expression, the middle one is assumed to be a query operator. If the query expression contains an even number of colons and is not a valid single DATETIME or INTERVAL value, INFORMIX-SQL returns an error.
..	An alternative search operator that specifies a range with DATETIME and INTERVAL data types. Because DATETIME and INTERVAL constants might include colons, use the . . search operator to avoid ambiguity.
?	A wildcard character. It represents a single character. A `?ick` search value in the First Name display field of the **orderform** form would find "Dick," "Rick," "Nick," and so on.

***** A wildcard character. It represents zero or more characters. An `S*` search value in the Last Name display field of the **orderform** form would find Sadler and Sipes. An `*r` search value would find five names: Baxter, Jaeger, Miller, Sadler, and Vector.

>> The highest-value search operator. Enter it (with no search value) in a display field to find the highest value for the field.

<< The lowest-value search operator. Enter it (with no search value) in a display field to find the lowest value for the field.

Usage

- Because of the way a computer stores floating-point numbers, you might not be able to retrieve FLOAT and SMALLFLOAT data by querying for the exact value you entered. You can solve this problem by using a range query. Specifying a FORMAT with a few places to the right of the decimal point in the FORMBUILD ATTRIBUTES section might also help.
- If the RIGHT attribute is specified for a display field, you might have to use an asterisk in front of a search value. (RIGHT does not right-justify the search value after you enter it.)
- Although the literals in PICTURE specifications appear on the screen when you add and update data, they do not appear on the screen when you query. If you enter the wrong literal value, your search will not be successful. A COMMENTS entry in the form file can help you avoid this problem.

GLS

The evaluation of less than (<) and greater than (>) expressions that contain character arguments is dependent on GLS settings. Refer to Appendix C, "Global Language Support," and the *Informix Guide to GLS Functionality*.♦

Related Options

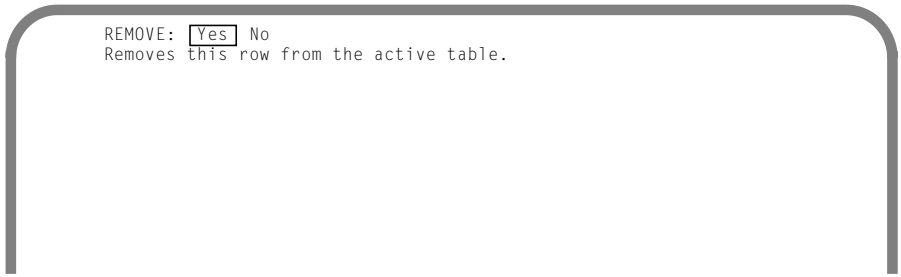
Next, Previous

REMOVE

Use the Remove option to delete the row on the screen from the active table.

Use the following procedure for the Remove option:

1. Use the Query, Next, and Previous options to display the row you want to delete.
2. Type r to select the Remove option.
3. PERFORM displays the following screen.



Enter y to delete the row, or n to keep it. In either case, the PERFORM menu appears on the screen next. The following message appears at the bottom of the screen when you remove a row:

```
Row deleted
```

Usage

You cannot remove a verify join row from one table (generally the master table, against which the join field is verified) unless you first remove all the rows that join it in other tables (generally detail tables, which are verified against the master table). For example, using the ORDERFORM form, you can remove rows in the **items** table. However, you cannot remove a row in the **orders** table without removing all rows in the **items** table that join that row because the Order Number display field is a verify join.

Related Options

Add, Update

SCREEN

Use the Screen option to cycle through the screen pages of the form.

Use the following procedure for the Screen option:

1. Type `s` to run the Screen option.
2. PERFORM displays the next screen page of the form. When you reach the last screen page, type `s` to display the first page again.

TABLE

Use the Table option when there is more than one table in the screen form and you want to select a new active table. Each table is assigned a table number assigned according to the order in which display field tags (including joins) for the table first appear in the ATTRIBUTES section of the form file. This number appears next to the table name or alias in the screen Information lines when the table is active. The Table option steps through the tables in table-number order starting with the active table.

Use the following procedure for the Table option:

1. Type `t`. If your screen form includes fields from two or more tables, PERFORM automatically selects and displays whichever page of the screen form contains the greatest number of fields from the new active table and surrounds those fields with delimiters (brackets); fields that belong to other tables do not have delimiters.

   If each table is on a separate screen page, PERFORM displays the screen page for the new active table.

2. Type `t` again. PERFORM displays the next sequential table. When you reach the last table, PERFORM displays the first table again.

Usage

If you know the number of the table you want to view next, you can go directly to that table without passing through the intervening tables. For example, suppose your form has five tables and you are looking at table number 4. If you want to see table number 2 next, type `2t` and PERFORM displays table number 2. Tables number 5 and 1 are skipped.

UPDATE

Use the Update option to modify the data in the displayed row of the current list.

Use the following procedure for the Update option:

1. Use the Query, Next, and Previous options to display the row you want to modify.
2. Type `u` to run the Update option. PERFORM puts the cursor in the first active field.
3. Edit the data, modifying as many display fields as you like.
4. Press ESCAPE to store the changed row, or the Interrupt key to ignore the changes and display the menu again.

Usage

- You cannot update a field that is a verify join field for another table without first updating the relevant field in the other table.
- If you press ESCAPE after you select the Update option, PERFORM displays the following message whether or not you actually changed anything:

```
This row has been changed.
```

- PERFORM sometimes introduces a slight discrepancy when you enter numbers in SMALLFLOAT or FLOAT fields. The entry `1.11`, for example, might display as `1.11000001` after you press RETURN or ESCAPE. This discrepancy occurs because of the way a computer stores numbers internally.

VIEW

Use the View option to display the contents of TEXT fields and of BYTE fields that are referenced in your form with the PROGRAM attribute. Blobs (Binary Large OBjects) include the TEXT and BYTE data types. You can only display the contents of the blob. You cannot change the blob from within the PERFORM form.

When you select the View option, INFORMIX-SQL positions the cursor on the first TEXT field, or the first BYTE field that uses the PROGRAM attribute. To display the blob, type an exclamation point (!). Press RETURN, TAB, or the down arrow key to skip to the next blob field that can be displayed; type an up arrow key to move to the previous blob field. Press ESCAPE to exit the View option and to redisplay the Main menu.

If you select the View option and the form contains no blob fields, INFORMIX-SQL displays the following error message:

```
There are no BLOB fields to view.
```

Chapter

4

The ACE Report Writer

In This Chapter

ACE is a general-purpose relational report writer that produces reports based on the tables of a database or the data in an ASCII input file. ACE can draw information from several database tables based on relationships that you specify among the tables when you design the report.

Creating and Compiling a Custom Report

You can create a report specification file based on a database table or tables in one of two ways:

- You can use the Report option on the ISQL (INFORMIX-SQL) Main menu.
- You can work directly with the appropriate programs from the operating-system command line.

Either alternative requires that you have already created the database and all the tables from which the report will draw information. The following two sections describe these alternative procedures. They do not, however, describe the rules for constructing or modifying the report specification file. These rules are defined in "Information About ACE" on page 4-10.

Creating a report from the command line is also described in Appendix G, "Accessing Programs from the Operating System." Use this option if you are retrieving data for the report from an input file.

Using the Menus to Create a Report

The procedure for creating, compiling, and running a report from the REPORT menu is described in the next two sections. "Generating a Default Report" explains the procedure used to produce a default report. "Creating a Custom Report" covers the steps involved in the production of a custom report. A more detailed description of each procedure is presented in the *INFORMIX-SQL User Guide*.

Generating a Default Report

To create a default report using the INFORMIX-SQL menu system, follow these steps:

1. Select the Report option on the INFORMIX-SQL Main menu and then the Report option on the REPORT menu.
2. If there is no current database, the CHOOSE DATABASE screen appears. After you select a database, the GENERATE REPORT screen is displayed. Enter the name you want to assign to the report (for example, **newrpt**). Do not use the **.ace** filename extension; INFORMIX-SQL automatically adds the required extension.
3. INFORMIX-SQL prompts you for the name of the table you want it to use to create the default report. After you enter the table name, INFORMIX-SQL automatically compiles the report specification and displays the REPORT menu. The report is now available to be used.
4. Select the Run option on the REPORT menu to run the report.

The default report specification file formats a report as a list of all columns in the table included in the report. It does not provide any special instructions to ACE about how to display the data, nor does it include instructions to perform data manipulations. Only one table contributes information to a default report.

Creating a Custom Report

To create a customized report using the INFORMIX-SQL menu system, follow these steps:

1. Complete the steps described in the previous section, "Generating a Default Report."
2. The REPORT menu should now appear on the screen. Select the Modify option on the REPORT menu.
3. The MODIFY REPORT screen appears. Enter the name of the default report (**newrpt**) just created.
4. If you have not specified an editor previously in the session or set the **DBEDIT** environment variable as described in Appendix B, "Setting Environment Variables," INFORMIX-SQL asks you to select the editor with which you want to work. Press RETURN if you want to select the editor whose name is displayed on the top line of the screen. If you want to work with a different editor, enter the name of the editor. INFORMIX-SQL calls the editor with the default report specification file.

   Modify the specification to include the data you need and the appearance you desire. Exit from the editor.
5. The MODIFY REPORT menu is displayed. Select the Compile option.
6. If your report specification file compiles correctly, a message to that effect is displayed, and ACE creates a report file with the filename extension **.arc** (for example, **newrpt.arc**). Go to step 8. If your report specification file contains errors, a message to that effect is displayed, and ACE creates a report file with the filename extension **.err** (for example, **newrpt.err**). Go to step 7.
7. Select the Correct option from the COMPILE REPORT menu. INFORMIX-SQL calls your system editor and the report specification file with the compiler errors. When you correct the errors, you need not delete the error messages. INFORMIX-SQL does that for you. Return to step 5.
8. When the compilation is successful, select the Save-and-exit option on the MODIFY REPORT menu. The REPORT menu is displayed. The report is now available for use.
9. Select the Run option on the REPORT menu and run the report.

As an alternative to using the Generate option and creating a default report specification, you can select the New option. INFORMIX-SQL calls your system editor, and you enter all the report specification instructions.

Creating a Report from the Command Line

To create a customized report specification directly from the operating-system command line, follow these steps:

1. Use the system editor to create a report specification file. Append the extension **.ace** to the filename.
2. Compile the specification with the ACEPREP program. Call ACEPREP as `saceprep`. You can omit the **.ace** filename extension when you call ACEPREP.

   For example, use this command line to compile the **newrpt.ace** specification file:

   ```
   saceprep newrpt
   ```
3. If the compilation is successful, ACE creates a compiled report file called **newrpt.arc** and you are finished creating your customized report. Go to step 5. If errors are detected in the report specification, a **newrpt.err** file is created. Go to step 4.
4. Use the system editor to edit this specification. Remove all error comments from the specification file. Overwrite the file **newrpt.ace** with this corrected version. Go to step 2.
5. To run the **newrpt.arc** report, use the ACEGO program. Call ACEGO as `sacego`. Do not include the .**arc** filename extension when you call ACEGO.

   For example, use this command line to run the **newrpt.arc** report:

   ```
   sacego newrpt
   ```

Creating a report from the command line is also described in Appendix G.

Command-Line Options

The following four command-line options are available for use with ACE:

-o Use the -o (output) option, followed by the pathname of a directory, to indicate the directory where ACEPREP places its output file. If you do not use this option, ACEPREP puts the file in your working (current) directory.

For example, to instruct ACEPREP to place the output file from compiling the NEWRPT specification in the OUTPUT directory, use the following command line:

```
saceprep -o output newrpt
```

-s Use the -s (silent) option with both ACEGO and ACEPREP to suppress all nonessential screen messages. For example, use this command line to suppress program banners in ACEPREP:

```
saceprep -s newrpt
```

-ansi When you use -ansi to compile a report, ACEPREP generates a warning whenever it encounters an Informix extension to the SELECT statement. ACEPREP places the warnings in a ***name*.err** file. When you invoke INFORMIX-SQL with **-ansi**, ACEPREP automatically checks for non-ANSI syntax. Use the following command:

```
saceprep -ansi newrpt
```

-d Use the -d (database) option with ACEGO, followed by the name of a database, to override the database that is named in the report specification. For example, to substitute the **sales** database for the database included in the NEWRPT specification, use this command line:

```
sacego -d sales newrpt
```

Do not use **-ansi** with ACEGO.

You can also check for non-ANSI syntax by setting the **DBANSIWARN** environment variable. See Appendix B for more information about using **DBANSIWARN**.

Information About ACE

The report specification file, the compiled report specification, and database files used in the report must be in your working directory or in a directory named in the **DBPATH** environment variable. You must refer to an input file by its full pathname if it is not in your current directory.

ACE Filename Conventions

ACE uses the following file-naming conventions:

- A report specification filename can be up to ten characters long. The filename must have an **.ace** filename extension. Without the **.ace** filename extension, the ACE compiler does not recognize the file.

  When you use the New or Generate options on the REPORT menu, INFORMIX-SQL automatically adds the **.ace** extension to the filename; you must not include it in the filename you choose.
- When you compile a report specification, you can omit the filename extension.
- The extension that the ACE compiler gives the output file depends on whether the compile is successful. If the compile is successful, the extension **.arc** is appended to the filename. If the compile is unsuccessful, the extension **.err** is appended to the filename.

  An **.err** file is a text file that contains the original specification file and error messages that describe and point to the problem ACE found.

Owner Naming

In an ANSI-compliant database, the prefix *owner.* must precede the table name if the report will be run by users other than the owner. The prefix *owner.* is optional in a database that is not ANSI-compliant. INFORMIX-SQL does check the accuracy of *owner.* if you include it in the statement, however.

Using Expressions in a Report Specification

An expression can be anything from a simple number or alphabetic constant to a more complex series of column values, functions, quoted strings, operators, and keywords. ACE evaluates expressions when it generates a report. It can display the result of the evaluation, assign it to a variable, or use it in a calculation.

When ACE evaluates an expression, it combines elements of the expressions that are separated from each other by operators. It combines elements in the order shown in Figure 4-1 on page 4-12. You can use parentheses to override this order.

When this manual refers to a number expression (*num-expr*), you can supply any type of expression, including character, as long as ACE can evaluate it as a number. The character string "123" is a valid number expression, while "m23" is not.

Similarly, *date-expr* is an expression that ACE can evaluate as a date. You can use a quoted string ("01012010" or "1-1-2010") or an INTEGER that evaluates to a legal date.

A quoted string is any string of characters in quotation marks. You can use a quoted string anywhere ACE requires a type CHAR or VARCHAR expression.

You cannot name a TEXT column in any arithmetic, aggregate, or Boolean expression, or in a BEFORE GROUP OF or AFTER GROUP OF clause.

You can name a TEXT column in a PRINT statement. The PRINT statement acts like a PRINT FILE statement with the TEXT item as a file.

Exponents are treated as integers and not as decimals. If a decimal is provided as an exponent, ACE truncates the number to an integer. For example, the expression `4 ** 3.4` is truncated to `4 ** 3` before it is evaluated.

Figure 4-1
Operator Precedence

Operator	Function	Precedence
-	unary minus	1
**	exponentiation	2
*	multiplication	3
/	division	4
+	addition	4
-	subtraction	4
is [not] null	presence / absence of a value	5
matches	equality for strings	5
=	equal	5
!= or <>	not equal	5
>	greater than	5
<	less than	5
>=	greater than or equal	5
<=	less than or equal	5
not	not	6
and	and	7
or	or	8

Precedence of operators: 1 is highest, 8 is lowest.

A unary minus indicates or changes the algebraic sign of a value (from positive to negative or from negative to positive). It operates on a single operand.

ACE Error Messages

The text of all INFORMIX-SQL error messages and suggestions for corrections is included in *Informix Error Messages* in Answer OnLine.

Sample Reports

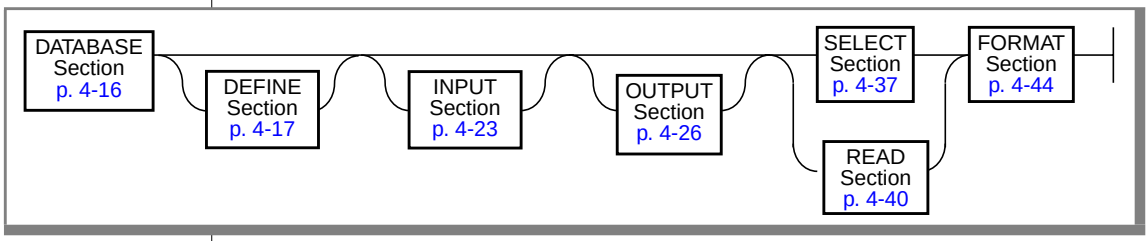
The sample reports in this chapter are taken from the following list. Additional reports, included with the demonstration database, are available for further study. These reports illustrate a variety of the commands available with ACE.

mail1.ace	A simple report that generates mailing labels
mail2.ace	A more sophisticated report that produces one column of mailing labels
mail3.ace	An interactive report that generates one to three columns of mailing labels
clist1.ace	A report that lists customer information
clist2.ace	An interactive customer report
ord1.ace	A custom report of orders placed with the store
ord2.ace	A second customer order report
ord3.ace	An interactive report that lists daily orders

Appendix A, "The Demonstration Database and Examples," contains the full text of each sample report specification.

Structure of a Report Specification File

A report specification file contains the instructions that specify what data a report includes and how that data appears. A report specification consists of three required sections (DATABASE, SELECT or READ, and FORMAT) and three optional sections (DEFINE, INPUT, and OUTPUT). The following diagram and list define the required and optional sections of a report specification.



The ACE report specification sections must be kept in the following general order:

1. DATABASE section: Each report specification must begin with a DATABASE section that identifies the database you want the report to use.
2. DEFINE section: The optional DEFINE section is used to declare variables that are used by the report as well as parameters that the report can accept from the command line. This section is also used to specify the field names and data types of values in an ASCII input file.
3. INPUT section: The INPUT section is optional. It is used to pass parameters to the report.
4. OUTPUT section: The OUTPUT section is optional. It is used to control page length and margin width, and to direct the output from the report to a file, a system printer, or a pipe.

5. SELECT or READ section:
   - SELECT section: If you retrieve data from a database table, the SELECT section specifies the columns and tables on which the report is based.
   - READ section: If you retrieve data from an ASCII file, the READ section specifies the name of the input file on which the report is based.
6. FORMAT section: The FORMAT section appears next. It includes commands that determine the appearance of the data in the report.

You can include comments anywhere in an ACE report specification. Simply enclose comments within a set of braces ({ }) or precede them with the pound sign (#) or double dash (--).

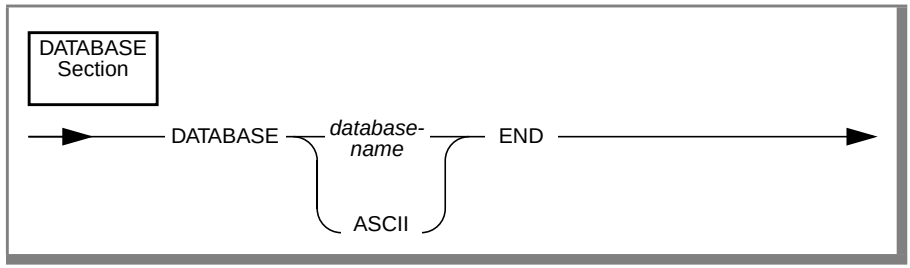
DATABASE Section

Every report specification must have a DATABASE section. The DATABASE section specifies the database ACE uses as the basis of the report. You can override the database that you specify in this section with the **-d** command-line option. See the section "Command-Line Options" on page 4-9 for more information.

The DATABASE section must be the first section in an ACE report specification. It begins with the DATABASE keyword, followed by the name of the database, and ends with the END keyword.

If you want to retrieve data from an ASCII file using the READ statement, you still must specify a database in the DATABASE section even though a report based on an ASCII file is not related to a database. You can either specify the name of an existing database or use the ASCII keyword.

The following diagram shows the structure of the DATABASE section.



database-name	is the name of the database accessed.

The following example DATABASE section is from the **clist1.ace** report:

```
database      {use stores7 database}
   stores7
end
```

The following example illustrates the use of the ASCII keyword:

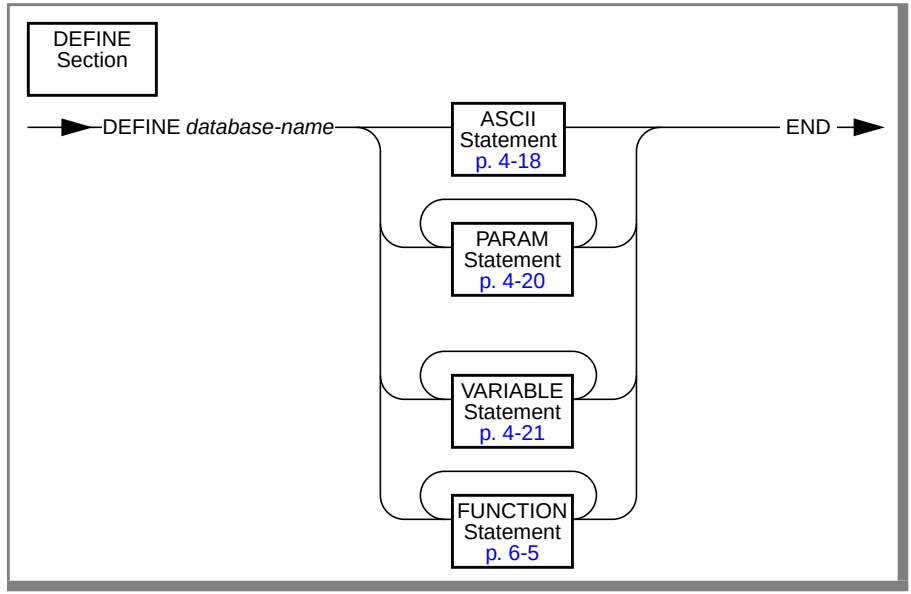
```
database      {using the READ statement}
   ascii
end
```

DEFINE Section

An ACE report specification can optionally contain a DEFINE section. The DEFINE section is used to declare variables used in the report and parameters the report can accept from the command line. If you are retrieving values from an input file using the READ statement, you must use the ASCII keyword in the DEFINE section to specify the field names and data types for the data in that file.

The DEFINE section begins with the DEFINE keyword and ends with the corresponding END keyword. The variable definition list appears between these keywords and is composed of one or more PARAM or VARIABLE statements or a combination of both. You can use a single ASCII keyword and field list between the DEFINE and END keywords.

The following diagram shows the structure of the DEFINE section.

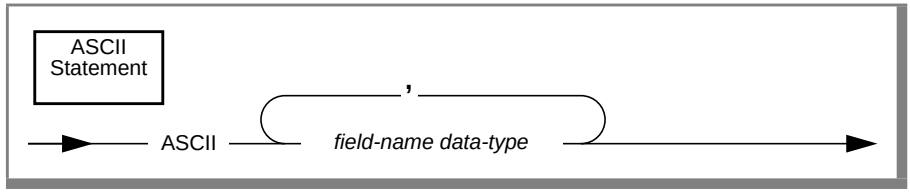


database-name	is the name of the database accessed.

The next three sections describe the ASCII, PARAM, and VARIABLE statements. The FUNCTION statement is described in Chapter 6, "Functions in ACE and PERFORM."

ASCII

You use an ASCII statement in a DEFINE section to specify the field names and data types of the records in an ASCII input file. The ACE report writer accesses this file in a READ statement.



ASCII	is a required keyword that specifies ASCII input.
field-name	is a required identifier for a field in the file.
data-type	is a valid SQL data type.

Usage

- You must include an ASCII statement in the DEFINE section if you use a READ statement in the READ section.
- You cannot use a SELECT statement to access an ASCII file, nor can you use a READ statement to access a database table.
- Although a report based on ASCII data is not related to a database, you must specify a database in the DATABASE section. Either specify the name of an existing database or use the ASCII keyword.
- The number of fields in the ASCII statement must match the number of fields in the ASCII file.
- Each *field-name* must be followed by a data type specification. ACE does not check the accuracy of data types, so run-time errors can occur if a data type has been specified incorrectly.
- No further specification of the MONEY data type is permitted beyond the keyword MONEY.
- See the *Informix Guide to SQL: Reference* for information about SQL data types.

The following ASCII statement defines a record from an ASCII file in unload format:

```
define
ascii stock_num smallint, manu_code char(3),
      description char(15), unit_price money,
      unit char(4), unit_descr char(15)
end
```

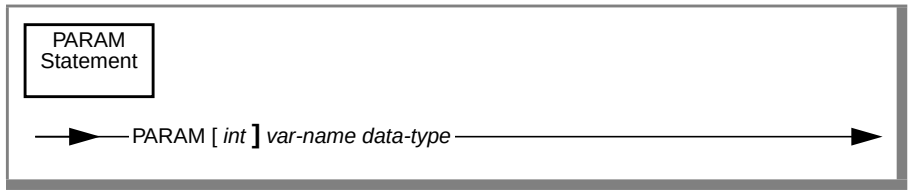
In this instance, the field names happen to have the same names and sequence as the column names in the **stock** table of the **stores7** database. Like the variable names of a PARAM or VARIABLE statement, the field names do not need to match the column names of any table. The number, order, and data types of the field names must be consistent with the fields in the ASCII file.

Related Commands

READ, UNLOAD

PARAM

This statement allows you to use arguments specified on the command line at the time you run an ACE report. It declares a variable whose initial value is that of a command-line argument. To use PARAM, you must call ACE from a custom user menu (see Chapter 5, "User-Menu") or the command line (see "Creating and Compiling a Custom Report" on page 4-5).



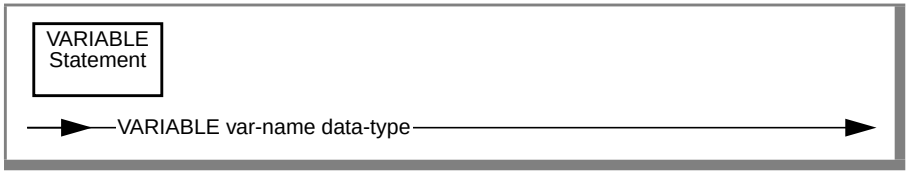
PARAM	is a required keyword.
int	is a required integer that specifies the position of the argument on the command line. The first argument is number 1.
var-name	is the name of the variable that you are declaring—it will initially have the value of a command-line argument when you run the report.
data-type	is a valid SQL data type.

Usage

- You can define a total of 100 variables using PARAM and VARIABLE statements in an ACE report specification.
- If a report specification uses a PARAM statement and you fail to provide arguments on the command line when you run the report, ACE gives an error message.
- If you want to use a variable defined by a PARAM statement in the SELECT section, you must precede the variable name with a dollar sign. Refer to the "SELECT Section" on page 4-37 for more information.
- See the *Informix Guide to SQL: Reference* for information about variable data types.

VARIABLE

This statement declares a variable that you can use in an ACE report specification.



VARIABLE	is a required keyword.
var-name	is the name of the variable that you are defining.
data-type	is a valid SQL data type.

Usage

- You can define a total of 100 variables using PARAM and VARIABLE statements in an ACE report specification.
- If you want to use a variable that you declare in a PARAM or VARIABLE statement in the SELECT section, you must precede the variable name with a dollar sign. (Refer to the "SELECT Section" on page 4-37 for more information.)
- No further specification of the MONEY data type is permitted beyond the keyword MONEY.
- VARCHAR columns and variables in expressions act the same way as CHAR columns and variables. When you define a VARCHAR in a report, do not give the *min-space* parameter with which the VARCHAR was defined for the database. Rather, indicate how many characters you want printed.

  For example, if in the **employee** table **history** is defined as VARCHAR(255,10), in the report you should define it as VARCHAR(255). If you only want to output a portion of the column, you can define VARCHAR with a shorter length, such as VARCHAR(120).
- See the *Informix Guide to SQL: Reference* for information about variable data types.

The following example is from the **ord3.ace** report:

```
define
   variable begin_date date
   variable end_date date
end
```

The following example shows the use of the VARCHAR data type:

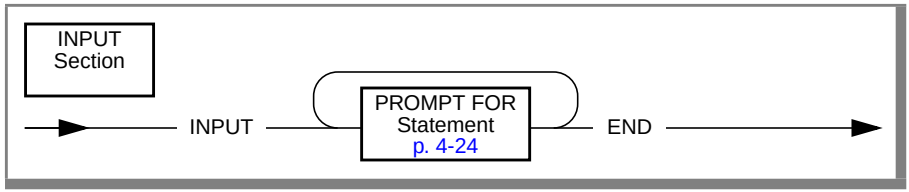
```
define
    variable history varchar(255)
end
```

The user enters the values for the variables when the program runs. See the "INPUT Section" on page 4-23 for a description of this process.

INPUT Section

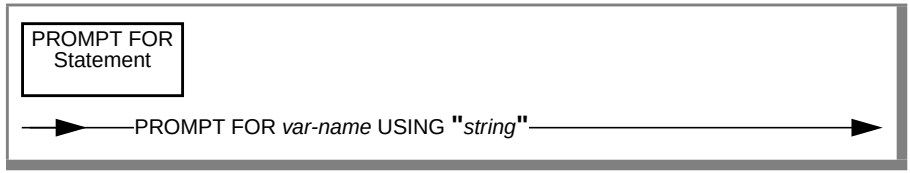
An ACE report specification optionally can contain an INPUT section. The INPUT section allows you to produce an interactive ACE report by prompting for and accepting input while ACE is running a report.

The INPUT section consists of the keywords INPUT and END with one or more PROMPT FOR statements in between. The following diagram shows the structure of the INPUT section.



PROMPT FOR

This statement prompts you while ACE is running a report and assigns the value you enter to a variable.



PROMPT FOR	are required keywords.
var-name	is the name of the variable that receives your input. You must declare this variable in the DEFINE section of the ACE report specification.
USING	is a required keyword.
string	is the string of characters that ACE uses as a prompt. You must enclose this string in quotation marks.

Usage

- You cannot prompt for, or accept, a database name using the PROMPT FOR statement. Refer to the DATABASE section and to the discussion of the **-d** option in "Command-Line Options" on page 4-9.
- You cannot prompt for, or accept, an output filename using the PROMPT FOR statement.

The following example is from the **ord3.ace** report:

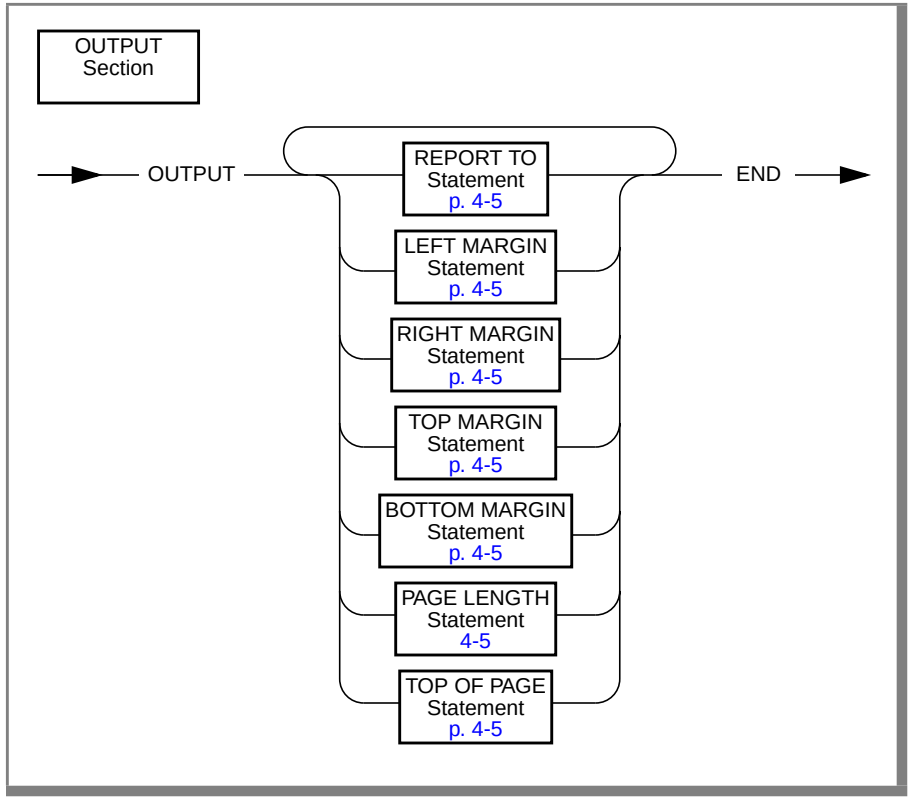
```
input
   prompt for begin_date
   using "Enter beginning date for report: "

   prompt for end_date
   using "Enter ending date for report: "
end
```

The two character strings `"Enter beginning date for report:"` and `"Enter ending date for report:"` appear as prompts on the screen when the **ord3.ace** report runs. The response to the first prompt is entered as the value to the *begin_date* variable; the response to the second prompt is entered as the value to the *end_date* variable. These two variables are used at several points in the **ord3.ace** report.

OUTPUT Section

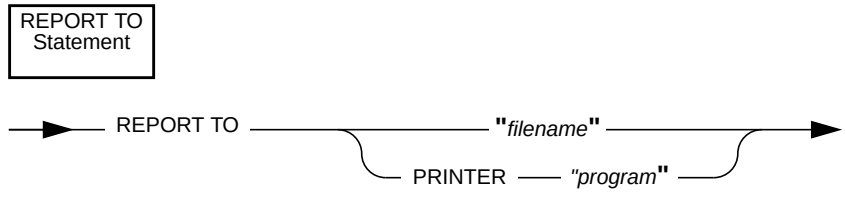
An ACE report specification can optionally contain an OUTPUT section. The OUTPUT section controls the width of the margins and the length of the page. The OUTPUT section also allows you to direct the output from the ACE report to a file or a printer.

The OUTPUT section begins with the OUTPUT keyword and ends with the corresponding END keyword, with one or more statements in between. The following diagram shows the structure of the OUTPUT section.



REPORT TO

This statement directs the output of the ACE report to a file or a printer.



REPORT TO	are required keywords.
filename	is the name of a system file that receives the report. You must enclose the filename in quotation marks.
PRINTER	is the keyword that sends the report to the printer.
program	is the name of a system command.

Usage

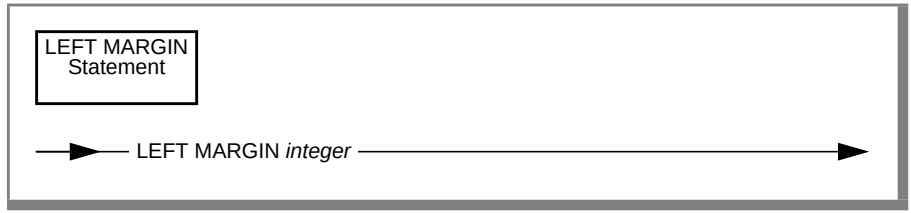
- When you do not use one of the REPORT TO statements, ACE sends the report to your screen.
- You cannot use more than one REPORT TO statement in a report specification.
- The TO PRINTER keywords cause ACE to send the report to the program named by the DBPRINT environment variable. If you do not define this environment variable, ACE sends the report to the **lp** program.
- If you want to send the report to a printer other than the system printer, you can use the REPORT TO *filename* statement to send the output to a file and then send the file to the printer of your choice.
- If the REPORT TO *filename* statement writes to an existing filename, the file is replaced with the new output. You can also use the REPORT TO PIPE statement to direct the output to a program that will send the output to the appropriate printer.

The following example directs the output to the **labels** file:

```
output
   report to "labels"
end
```

LEFT MARGIN

This statement sets a left margin for a report.



LEFT MARGIN	are required keywords.
integer	is an integer that specifies the width of the left margin in spaces.

Usage

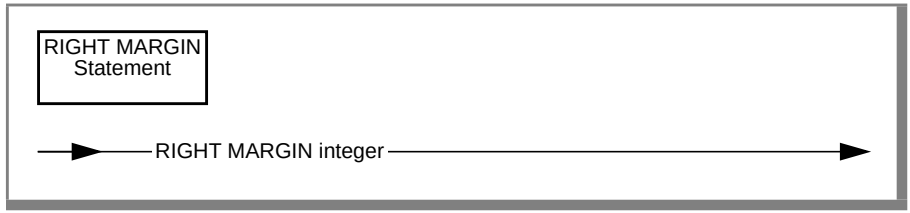
The default left margin is five spaces.

The following example is from the **mail2.ace** report. ACE prints the left side of the report as far to the left as possible.

```
output
   top margin 0
   bottom margin 0
   left margin 0
   page length 9
   report to "labels"
end
```

RIGHT MARGIN

This statement sets a right margin for a report.



RIGHT MARGIN	are required keywords.
integer	is an integer that specifies the width of the text on the page in characters.

Usage

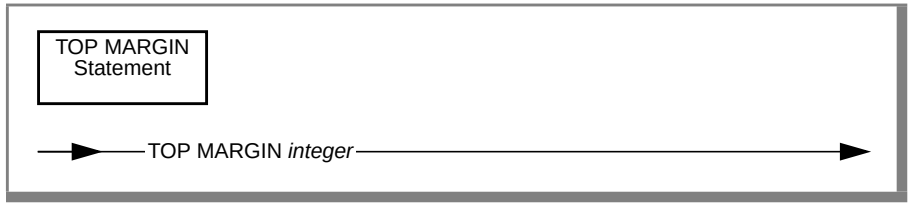
- The RIGHT MARGIN determines the right margin by specifying the width of the page in characters. It does not depend on the LEFT MARGIN but always starts its count from the left edge of the page (space 0).
- The RIGHT MARGIN is only effective when the FORMAT section contains an EVERY ROW statement.
- The default right margin is 132 characters.

The following example report specification demonstrates the use of the RIGHT MARGIN statement. ACE sets the right margin for the report at 70 characters.

```
database
   stores7
end

output
   right margin 70
end

select *
   from customer
end

format
   every row
end
```

TOP MARGIN

This statement sets a top margin for a report.



TOP MARGIN	are required keywords.
integer	is an integer that specifies the number of blank lines that ACE leaves at the top of each page.

Usage

- The default top margin is three lines.
- The top margin appears above any page header you specify.

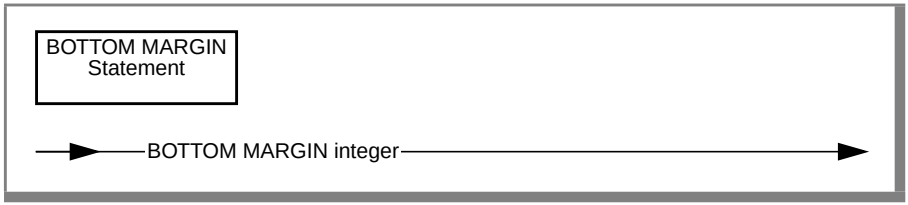
Example

The following example is from the **mail2.ace** report. ACE begins printing at the top of each page.

```
output
   top margin 0
   bottom margin 0
   left margin 0
   page length 9
   report to "labels"
end
```

BOTTOM MARGIN

This statement sets a bottom margin for a report.



BOTTOM MARGIN	are required keywords.
integer	is an integer that specifies the number of blank lines that ACE is to leave at the bottom of each page.

Usage

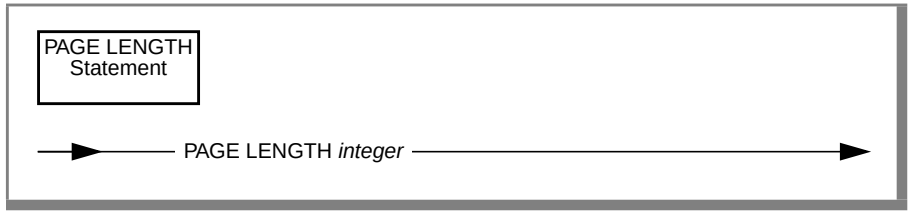
- The default bottom margin is three lines.
- The bottom margin appears below any page trailer.

In the following example, the printing continues to the bottom of each page:

```
output
   top margin 0
   bottom margin 0
end
```

PAGE LENGTH

This statement sets the number of lines on each page of a report.



PAGE LENGTH	are required keywords.
integer	is an integer that specifies the length of the page in lines.

Usage

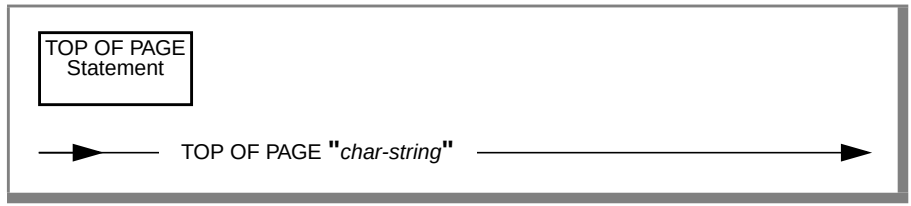
- The default page length is 66 lines.
- The PAGE LENGTH includes the TOP MARGIN and BOTTOM MARGIN.

The following example demonstrates the use of the PAGE LENGTH statement. ACE prints each page with 22 lines.

```
output
   {This length works on std 24-line crt}
   page length 22
   top margin 0
   bottom margin 0
end
```

TOP OF PAGE

This statement specifies the character string that causes your printer to eject a page.



TOP OF PAGE	are required keywords.
char-string	is a one- or two-character string that causes your printer to eject a page.

Usage

- On most printers, *char-string* is "^L", the ASCII form-feed character. ACE uses the first character of the string as the TOP OF PAGE character unless it is the ^ character. If the first character is the ^ character, ACE decodes the second character as a control character. (If you are unsure of the character string to specify for your printer, refer to the documentation provided with your printer.)
- ACE places the character string in the report to advance to the next page whenever the program causes a new page to be set up. Any of the following items can initiate a new page:
  - The next print line meets the bottom margin.
  - A SKIP TO TOP OF PAGE statement is executed.
  - A SKIP *n* LINES statement skips more lines than are available on the current page.
  - A NEEDS statement specifies more lines than are available on the current page.
- If you specify the TOP OF PAGE statement, ACE uses the specified page-eject character to set up new pages instead of using line feeds.
- If you omit the TOP OF PAGE statement, ACE fills the remaining lines of the current page with line feeds when a new page is set up.

The following example sets CONTROL-L as the page-eject character:

```
output
   top of page "^L"
   report to "r_out"
end

select * from customer
end

format
   on every row
end
```

SELECT Section

Every report specification must have a SELECT section or a READ section. The SELECT section specifies the columns or tables or both that the report is based on if you retrieve data from a database. The READ section specifies the input file the report is based on if you retrieve data from an ASCII file. See the "READ Section" on page 4-40 for details on how to use ASCII files in ACE reports.

You can use the SELECT section to specify criteria for selecting and ordering rows based on the contents of specific columns. The FORMAT section can group rows in the report based on the order you specify in the SELECT section.

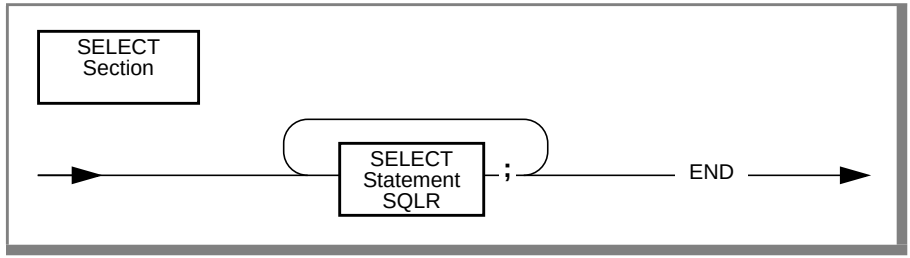
The SELECT section contains one or more SELECT statements. These statements are identical to the SELECT statements described in the *Informix Guide to SQL: Syntax*. This chapter does not define the SELECT statement but shows how to incorporate it in an ACE report specification.

The SELECT section begins with the SELECT keyword. This keyword introduces both the SELECT section *and* the first SELECT statement. Other SELECT statements can follow the first—each must begin with the SELECT keyword. All SELECT statements, except for the last, must end with a semicolon. (If there is only one SELECT statement, it does not require a semicolon.) The SELECT section ends with the END keyword. All but the last SELECT statement must have an INTO TEMP clause.

If you use an ORDER BY clause in the SELECT section, you cannot use an integer or a column with a table prefix (*table.column*) to indicate the column to sort by. If you cannot use the column name alone because it is not unique or because it is an expression, define a display label in the select list and use it in both the ORDER BY clause and the FORMAT section in the AFTER and BEFORE GROUP OF control blocks. The second example in this section demonstrates the use of a display label.

If you use an ACE variable in the SELECT section, you must precede the variable name with a dollar sign. The third example in this section demonstrates the use of a variable name.

The following diagram shows the structure of the SELECT section.



The following example is from the **mail1.ace** report. ACE selects all rows from the **customer** table and orders the rows first by zip code and then by last name.

```
select *
   from customer
   order by zipcode, lname
end
```

The following example is from the **ord1.ace** report:

```
select
   orders.order_num number,
   order_date, customer_num,
   po_num, ship_date, ship_charge,
   paid_date,

   items.order_num, stock_num, manu_code,
   quantity, total_price

   from orders, items

   where orders.order_num = items.order_num

   order by number
end
```

ACE selects the indicated columns from the **orders** and **items** tables. The **order_num** column in the **orders** table is given the display label **number** and is joined to the **order_num** column in the **items** table. ACE orders the rows by the values in the **number** column.

The following example is from the **clist2.ace** report:

```
select
      customer_num,
      fname,
      lname,
      company,
      city,
      state,
      zipcode,
      phone
   from
      customer
   where
      state matches $thisstate
  order by
      zipcode,
      lname
end
```

ACE selects the indicated columns from the **customer** table. The WHERE clause tells ACE to select only those rows where the value in the **state** column matches the value in the variable **thisstate**. ACE orders the rows by the values in the **zipcode** column first and then by **lname**.

READ Section

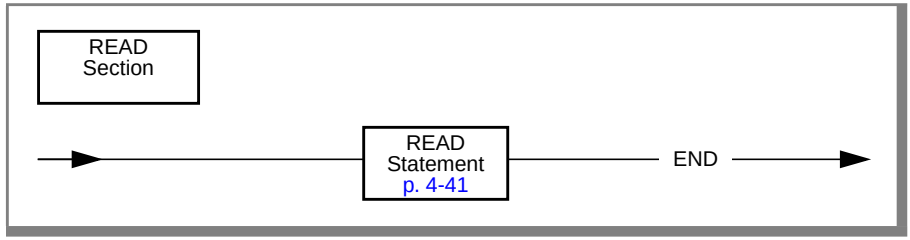
As an alternative to the SELECT section, you can include a READ section containing a READ statement. Unlike the SELECT statement, which queries the database for rows, the READ statement retrieves rows from an ASCII input file. Every report specification must have either a READ section or a SELECT section.

The READ statement allows you to retrieve data from ASCII files produced by the UNLOAD statement of SQL or the Output option of PERFORM. In addition, you can produce reports from ASCII files created or edited by other software products.

The following conditions must be satisfied before you can read data from an ASCII file:

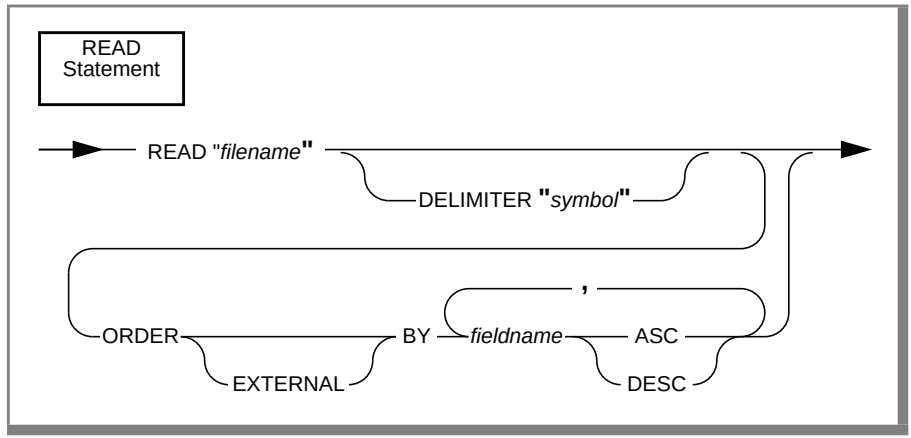
- You must know the complete pathname of the ASCII data file. You must also know the number of fields, the delimiter symbol, and the order and data type of each field of a record in the file.
- You must use the ASCII statement in the DEFINE section of a report specification to indicate the format of a record in the ASCII file. The ASCII keyword is followed by an ordered list of the field names and data types of the ASCII file. See the "DEFINE Section" on page 4-17 for details on using the ASCII statement.
- Although a report based on ASCII data is not related to a database, you must include a DATABASE section in the report specification. You can either specify an existing database or use the ASCII keyword. See the "DATABASE Section" on page 4-16 for details on using the ASCII keyword.

The following diagram shows the structure of the READ section.



READ

Use the READ statement in the READ section to retrieve data from an ASCII input file in unload format. The READ section specifies the name of the input file, any nondefault delimiter, and optional sorting specifications. You use the ASCII statement in the DEFINE section to specify the fields of each record in the input file.



READ	is a required keyword that specifies ASCII input.
filename	is the name of the ASCII file enclosed in quotes. If the file is not located in the current directory, you must include the complete pathname.
DELIMITER	is an optional keyword that is used if the field separator symbol is not the default delimiter (|) .
symbol	is the character, enclosed in quotation marks, that is used between fields of the records in filename.
ORDER BY	is an optional keyword that is required if records from the input file are to be sorted in the report according to the values in one or more fields.
EXTERNAL	is an optional keyword indicating that the records in the input file are already sorted.

fieldname	is the name of a field in an input file record, as defined in the ASCII statement of the DEFINE section, that is used as a sorting key.
ASC	is an optional keyword specifying that the values in the *fieldname* field are used to sort records in ascending order (smallest values first).
DESC	is an optional keyword specifying that the values in the *fieldname* field are used to sort records in descending order (largest values first).

Usage

- The READ statement requires an ASCII statement in the DEFINE section. You cannot use a SELECT statement to access an ASCII file, nor can you use a READ statement to access a database table.
- The default delimiter is a vertical bar (| = ASCII 124). See Appendix B, "Setting Environment Variables," for information on how to specify a different default delimiter with the DBDELIMITER environment variable.
- ACE uses the delimiter specified in the READ statement as the field separator, regardless of whether you have set the DBDELIMITER environment variable.
- An ORDER BY clause in a READ statement can list up to eight field names as sorting keys. These names must match the field names that are specified in the ASCII statement. Specify an existing database, rather than use the ASCII keyword, if you are using an ORDER BY clause.
- If more than one sorting key is specified in an ORDER BY clause, the primary key is the first field named in the ORDER BY list, the secondary sorting key is the second field named in that list, and so on.
- If the ASCII file named in a READ statement is already sorted, and the FORMAT section contains BEFORE GROUP OF or AFTER GROUP OF control blocks on two or more fields, then an ORDER EXTERNAL BY clause must specify the hierarchy of the order.
- ACE does not allow the use of the space or double quotation mark (") as a delimiter in the ASCII file.

The following READ statement specifies an ASCII file (in unload format) that corresponds to the **stock** table of the **stores7** database:

```
read "stock1" delimiter ":"
   order by unit_price desc, description
end
```

This READ statement tells ACE to read the records in an ASCII file called STOCK1 that uses the colon (:) as a delimiter, and to sort the records in descending order according to the values in the field **unit_price**. Since the **description** field is a secondary sorting key, records that have the same **unit_price** value appear in ascending alphabetical order according to the label in their **description** field.

The next example reverses the previous order of the sorting keys and sorts unit prices in default (ascending) order:

```
read "stock1" delimiter ":"
   order external by description, unit_price
end
         . . .
format
         . . .
after group of description
         . . .
after group of unit_price
         . . .
```

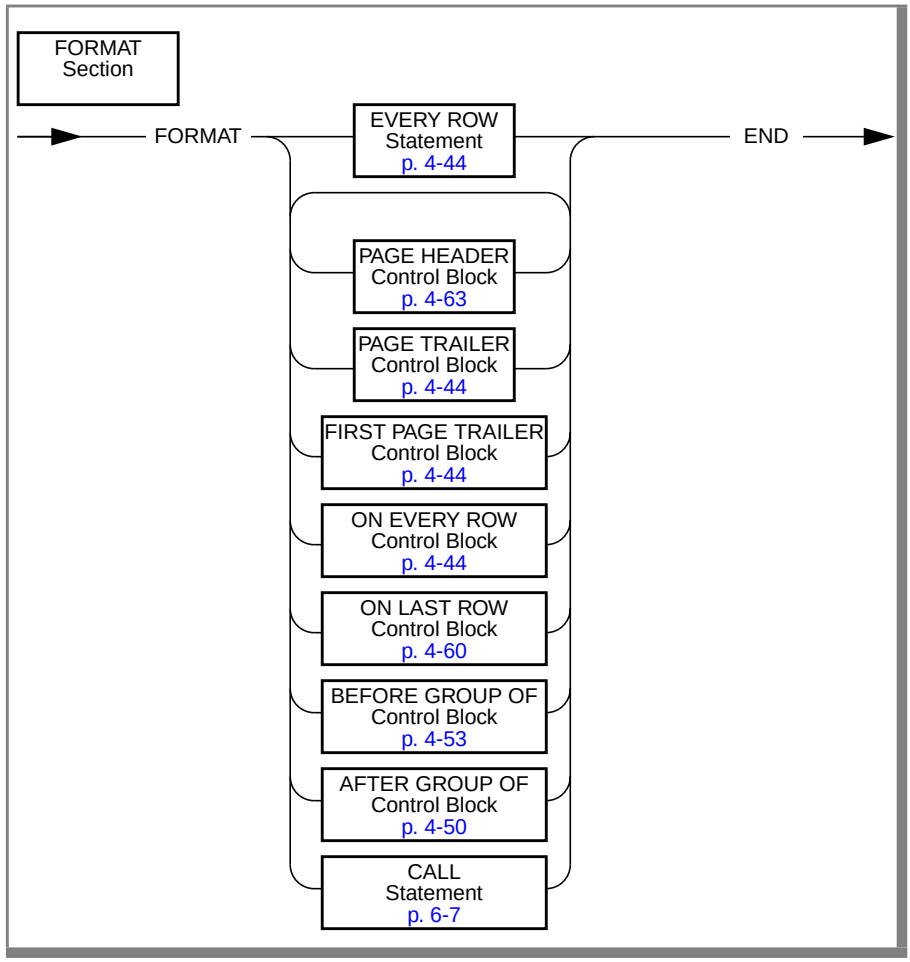
The ORDER EXTERNAL BY clause specifies that the **stock1** file is already sorted. Totals or subtotals specified in the AFTER GROUP OF control blocks are printed after the groups of rows have been printed according to the sorting instructions in the ORDER BY clause.

Related Commands

ASCII, ORDER BY (option of SELECT), OUTPUT (option of PERFORM), UNLOAD (SQL statement)

FORMAT Section

An ACE report specification must contain a FORMAT section. The FORMAT section determines the appearance of the report. It works with the data that is qualified by the last (or only) SELECT statement in the SELECT section, or with the contents of an ASCII file referenced by the READ statement in the READ section. The FORMAT section begins with the FORMAT keyword and ends with the corresponding END keyword as shown in the following diagram.



The simplest FORMAT section contains only an EVERY ROW statement between the FORMAT and END keywords. If you use an EVERY ROW statement, you cannot use any other statements or control blocks in the FORMAT section. The following example shows the structure of this type of FORMAT section:

```
FORMAT
   EVERY ROW statement
END
```

More complex FORMAT sections can contain control blocks such as ON EVERY ROW and BEFORE GROUP OF. Each of these control blocks must contain at least one statement such as PRINT or SKIP *n* LINES. In the FORMAT section, you cannot refer to a column using its table name (*table.column*) as a prefix. If you do not use an EVERY ROW statement, you can combine control blocks as required. You can place control blocks in any order within the FORMAT section.

EVERY ROW

The EVERY ROW statement causes ACE to output every row that the SELECT or READ section retrieves. It uses a default format.

Usage

- This statement is useful when you want to develop a report quickly using a default format. The report uses as column headings the column names you assigned when you created the table or the field names that you assigned in the ASCII statement. Because the EVERY ROW statement cannot contain any control blocks or other statements, you cannot alter the default format to create a custom report.
- The EVERY ROW statement stands by itself—you cannot modify it with any of the statements listed in "Statements" on page 4-65.
- When you use the EVERY ROW statement, you cannot use any control blocks in the FORMAT section.
- A report generated by an EVERY ROW statement uses the column names you assigned when you created the table.
- If the columns that you specify in the SELECT section, or the values that you retrieve in the READ section, fit on one line, ACE produces a report with column or field names across the top of each page; otherwise, ACE produces a report with the column or field names down the left side of the page.
- You can use the RIGHT MARGIN statement in the OUTPUT section to control the width of a report that uses the EVERY ROW statement.
- Use the ON EVERY ROW control block if you want to display every row in a format other than the default format. (See the discussion of "ON EVERY ROW" on page 4-58.)

The following example shows a minimal ACE report specification using the EVERY ROW statement:

```
database
   stores7
end

select *
   from customer
end

format
   every row
end
```

The following example shows a portion of the output from the preceding specification:

```
customer_num  101
fname         Ludwig
lname         Pauli
company       All Sports Supplies
address1      213 Erstwild Court
address2
city          Sunnyvale
state         CA
zipcode       94086
phone         408-791-8075

customer_num  102
fname         Carole
lname         Sadler
company       Sports Spot
address1      785 Geary St
address2
city          San Francisco
state         CA
zipcode       94117
phone         415-822-1291

customer_num  103
fname         Philip
lname         Currie
      .
      .
      .
```

The following example shows another example of a brief report specification that uses the EVERY ROW statement:

```
database
   stores7
end

select order_num, customer_num,
   order_date
   from orders
end

format
   every row
end
```

The following example shows the output from the preceding specification:

```
order_num customer_num order_date

     1001          104  01/20/1991
     1002          101  06/01/1991
     1003          104  10/12/1991
     1004          106  04/12/1991
     1005          116  12/04/1991
     1006          112  09/19/1991
     1007          117  03/25/1991
     1008          110  11/17/1991
     1009          111  02/14/1991
     1010          115  05/29/1991
     1011          104  03/23/1991
     1012          117  06/05/1991
     1013          104  09/01/1991
     1014          106  05/01/1991
     1015          110  07/10/1991
```

Control Blocks

Control blocks provide the structure for a custom report. The control blocks that you can use in a FORMAT section follow:

- AFTER GROUP OF
- BEFORE GROUP OF
- FIRST PAGE HEADER
- ON EVERY ROW
- ON LAST ROW
- PAGE HEADER
- PAGE TRAILER

Each control block is optional, but if you do not use the EVERY ROW statement, you must include at least one control block in a report specification.

Each control block must include at least one statement. (See "Statements" on page 4-65.) If you have INFORMIX-ESQL / C, you can also call C functions from within a control block. See the *INFORMIX-ESQL/C Programmer's Manual* and Chapter 6, "Functions in ACE and PERFORM," for details.

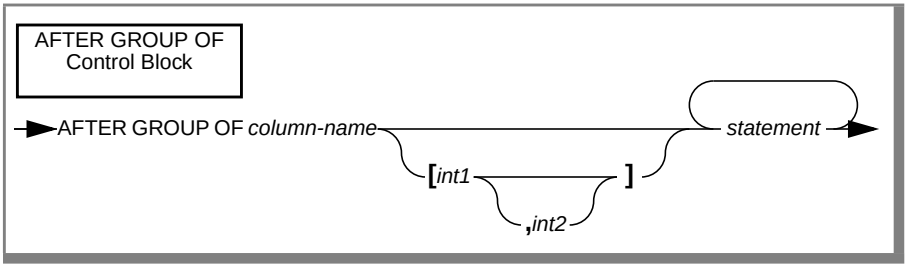
When you use an ORDER BY clause in the SELECT or READ section of an ACE report specification, you can use BEFORE GROUP OF and AFTER GROUP OF control blocks in the FORMAT section. When you use the BEFORE GROUP OF, AFTER GROUP OF, and ON EVERY ROW control blocks in a single report specification, ACE processes the control blocks in the order shown in Figure 4-2. (The figure assumes that the SELECT or READ section orders by columns **a**, **b**, and **c**.)

Figure 4-2
Order of Group Processing

```
before group of a
    before group of b
        before group of c
            on every row
        after group of c
    after group of b
after group of a
```

AFTER GROUP OF

The AFTER GROUP OF control block specifies what action ACE takes after it processes a group of rows. Grouping is determined by the ORDER BY clause of the SELECT or READ section.



AFTER GROUP OF	are required keywords.
column-name	is the name of one of the columns or identifiers specified in the ORDER BY clause of the SELECT or READ section.
int1,*int2*	are optional subscripts that you can use with a type CHAR column to refer to a subset of the column beginning at character position *int1* and ending at character position *int2*. If *int2* is not specified, the end of the column is used.
statement	is a list of one or more statements or a compound statement.

Usage

- A *group* of rows is all the rows that contain the same value for a given column. ACE automatically groups rows when you use an ORDER BY clause in the SELECT or READ section of a report specification (that is, groups come together when you order a list).

  When you specify more than one column in an ORDER BY clause, ACE orders the rows first by the first column you specify (most significant), second by the second column you specify, and so on, until the last column you specify (least significant).

  ACE processes the statements in an AFTER GROUP OF control block each time the specified column changes value, each time a more significant column changes value, and at the end of a report. (See Figure 4-2 on page 4-49.)

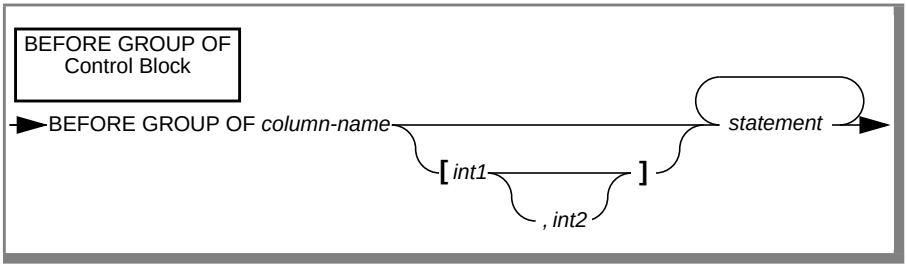
- Each column specified in the ORDER BY clause in the SELECT or READ section can contain one AFTER GROUP OF control block.
- If you have more than one AFTER GROUP OF control block, their order within the FORMAT section is not significant. ACE processes the AFTER GROUP OF control blocks in the reverse order specified in the ORDER BY clause in the SELECT or READ section. (See Figure 4-2 on page 4-49.)
- When ACE finishes generating a report, it executes all of the statements in the AFTER GROUP OF control blocks before it executes those in the ON LAST ROW control block.
- You cannot reference the *column-name* in the AFTER GROUP OF clause using the name of a table (the *table.column* structure is not allowed). If you cannot use the column name alone because it is not unique or because it is an expression, define a display label in the select list of the SELECT section and use it in the AFTER GROUP OF clause.
- You cannot use an integer to indicate by which column of the select list the grouping is to occur.
- You can only use group aggregates in AFTER GROUP OF control blocks. You cannot use group aggregates in any other control blocks.
- If you specify a substring of a CHAR column in an ORDER BY clause in the SELECT or READ section, you must use the same substring specification as the *column-name* in an AFTER GROUP OF control block.
- You can use a SKIP TO TOP OF PAGE statement in an AFTER GROUP OF control block to start a new page after each group.
- When ACE processes the statements in an AFTER GROUP OF control block, the columns that the report is processing still have the values from the last row of the group. From this perspective, the AFTER GROUP OF control block could be called the ‘‘on last row of group’’ control block.

The following example is from the **ord2.ace** report:

```
after group of number

skip 1 line
print 4 spaces, "Shipping charges for the order:  ",
    ship_charge using "$$$$.&&"
skip 1 line

print 5 spaces, "Total amount for the order:  ",
   ship_charge + group total of total_price
   using "$$,$$$,$$$.&&"
skip 3 lines

after group of custnum
skip 2 lines
```

BEFORE GROUP OF

The BEFORE GROUP OF control block specifies what action ACE is to take before it processes a group of rows. Grouping is determined by the ORDER BY clause of the SELECT or READ section.



BEFORE GROUP OF	are required keywords.
column-name	is the name of one of the columns or identifiers specified in the ORDER BY clause of the SELECT or READ section.
int1,*int2*	are optional subscripts that you can use with a type CHAR column to refer to a subset of the column beginning at character position *int1* and ending at character position *int2*. If *int2* is not specified, the end of the column is used.
statement	is a list of one or more statements or a compound statement.

Usage

- A *group* of rows is all the rows that contain the same value for a given column. ACE automatically groups rows when you use an ORDER BY clause in the SELECT or READ section of a report specification (that is, groups come together when you order a list).

  When you specify more than one column in an ORDER BY clause, ACE orders the rows first, by the first column you specify (most significant), second, by the second column you specify, and so on, until the last column you specify (least significant).

  ACE processes the statements in a BEFORE GROUP OF control block at the start of a report, each time the specified column changes value, and each time a more significant column changes value. (See Figure 4-2 on page 4-49.)

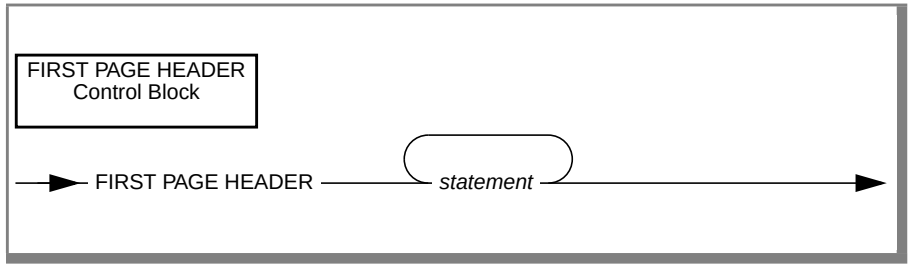
- Each column specified in the ORDER BY clause in the SELECT or READ section can contain one BEFORE GROUP OF control block.
- If you have more than one BEFORE GROUP OF control block, their order within the FORMAT section is not significant. ACE processes the BEFORE GROUP OF control blocks in the reverse order specified in the ORDER BY clause in the SELECT or READ section. (See Figure 4-2 on page 4-49.)
- When ACE starts to generate a report, it executes all of the statements in the BEFORE GROUP OF control blocks before it executes those in the ON EVERY ROW control block.
- You cannot reference the *column-name* in the BEFORE GROUP OF clause using the name of a table (the *table.column* structure is not allowed). If you cannot use the column name alone because it is not unique or because it is an expression, define a display label in the select list of the SELECT section and use it in the BEFORE GROUP OF clause.
- You cannot use an integer to indicate by which column of the select list the grouping is to occur.
- If you specify a substring of a CHAR column in an ORDER BY clause in the SELECT or READ section, you must use the same substring specification as the *column-name* in a BEFORE GROUP OF control block.
- You can use a SKIP TO TOP OF PAGE statement in a BEFORE GROUP OF control block to start a new page after each group.
- When ACE processes the statements in a BEFORE GROUP OF control block, the columns that the report is processing have the values from the first row of the row group. From this perspective, the BEFORE GROUP OF control block could be called the “on first row of group” control block.
- You cannot name a TEXT column in a BEFORE GROUP OF or AFTER GROUP OF clause.

The following example is from the **ord1.ace** report:

```
before group of ordnum

print "Order number:  ", ordnum using "#####",
   " for customer number:  ", customer_num
   using "#####"
print "Customer P.O. :  ", po_num,
   "  Date ordered:  ", order_date

skip 1 line
print "Stockno", column 20,
   "Mfcode", column 28, "Qty", column 38, "Price"
```

FIRST PAGE HEADER

The FIRST PAGE HEADER control block specifies what information appears at the top of the first page of the report.



FIRST PAGE HEADER	are required keywords.
statement	is a list of one or more statements or a compound statement.

Usage

- The vertical size of the first page header is equal to the number of lines that you specify in the FIRST PAGE HEADER control block. The TOP MARGIN (in the OUTPUT section) affects how close to the top of the page ACE displays the page header.
- A FIRST PAGE HEADER control block overrides a PAGE HEADER control block on the first page of a report.
- You cannot use the SKIP TO TOP OF PAGE statement in a FIRST PAGE HEADER control block.
- If you use an IF THEN ELSE statement in a FIRST PAGE HEADER control block, the number of lines displayed by the PRINT and SKIP statements following the THEN keyword must be equal to the number of lines displayed by the PRINT and SKIP statements following the ELSE keyword.
- You cannot use the PRINT FILE statement to read and display text from a file in a FIRST PAGE HEADER control block.
- You can use a FIRST PAGE HEADER control block to produce a title page as well as column headings.

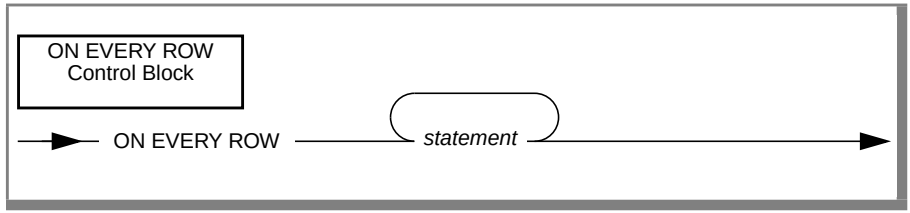
The following example is from the **mail3.ace** report:

```
first page header

let i = 1

let l_size = 72/count1

let white = 8/count1
```

This FIRST PAGE HEADER does not display any information. Because ACE executes the FIRST PAGE HEADER control block *before* it generates any output, you can use this control block (as demonstrated in the example) to initialize variables that you use in the FORMAT section.

ON EVERY ROW

The ON EVERY ROW control block specifies what action ACE takes after the SELECT section qualifies a row, or after the READ section retrieves a row.



ON EVERY ROW	are required keywords.
statement	is a list of one or more statements or a compound statement.

Usage

- ACE processes the statements in an ON EVERY ROW control block as each new row is formatted.
- If a BEFORE GROUP OF control block is triggered by a change in column value, all BEFORE GROUP OF control blocks are executed (in the order of their significance) before the ON EVERY ROW control block is executed.
- If an AFTER GROUP OF control block is triggered by a change in column value, all AFTER GROUP OF control blocks are executed (in the order of their significance) after the ON EVERY ROW control block is executed.
- You cannot name a TEXT column in a BEFORE GROUP OF or AFTER GROUP OF clause.

The following example is from the **clist1.ace** report:

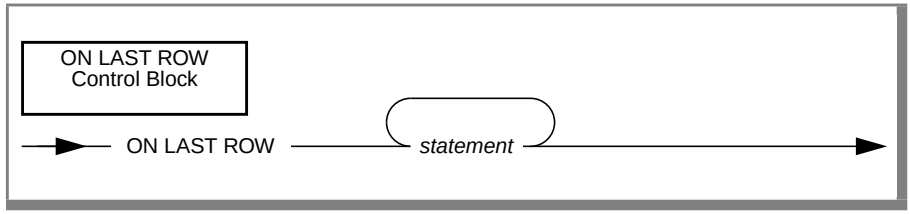
```
on every row
    print customer_num using "####",
    column 9, fname clipped, 1 space, lname clipped,
    column 32, city clipped, ", " , state,
    column 54, zipcode,
    column 62, phone
```

The following example is from the **mail2.ace** report:

```
on every row
   if (city is not null) and
      (state is not null) then
   begin
      print fname clipped, 1 space, lname
      print company
      print address1
      if (address2 is not null) then
         print address2
      print city clipped, ", " , state,
      2 spaces, zipcode
      skip to top of page
   end
```

ON LAST ROW

The ON LAST ROW control block specifies the action ACE takes after processing the last row qualified by the SELECT section, or the last row retrieved by the READ section.



ON LAST ROW	are required keywords.
statement	is a list of one or more statements or a compound statement.

Usage

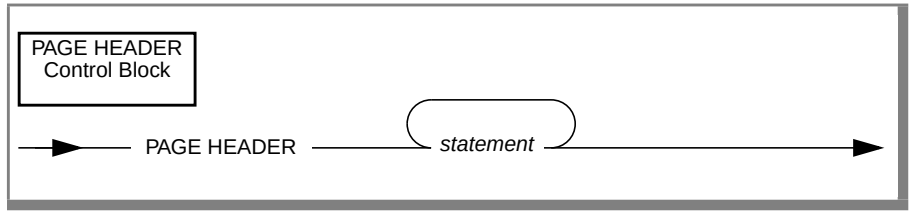
- ACE executes the statements in the ON LAST ROW control block after it executes those in the ON EVERY ROW and AFTER GROUP OF control blocks.
- You can use the ON LAST ROW control block to display report totals.
- When ACE processes the statements in an ON LAST ROW control block, the values from the last row returned by the SELECT statement or the last row retrieved by the READ statement are current and can be used.

The following example is from the **clist1.ace** report:

```
on last row
    skip 1 line
    print  "TOTAL NUMBER OF CUSTOMERS:",
        column 30, count using "##"
```

PAGE HEADER

The PAGE HEADER control block specifies what information will appear at the top of each page of the report.



PAGE HEADER	are required keywords.
statement	is a list of one or more statements or a compound statement.

Usage

- The vertical size of the page header is equal to the number of lines that you specify in the PAGE HEADER control block. The TOP MARGIN (in the OUTPUT section) affects how close to the top of the page ACE displays the page header.
- A FIRST PAGE HEADER control block overrides a PAGE HEADER control block on the first page of a report.
- You cannot use the SKIP TO TOP OF PAGE statement in a PAGE HEADER control block.
- If you use an IF THEN ELSE statement in a PAGE HEADER control block, the number of lines displayed by the PRINT and SKIP statements following the THEN keyword must be equal to the number of lines displayed by the PRINT and SKIP statements following the ELSE keyword.
- If you use a FOR or WHILE statement that contains a PRINT statement in a PAGE HEADER control block, you must terminate the PRINT statement with a semicolon. The semicolon suppresses any NEWLINE (RETURN) characters in the loop, keeping the number of lines in the header constant from page to page.
- You cannot use a PRINT FILE statement to read and display text from a file in a PAGE HEADER control block.

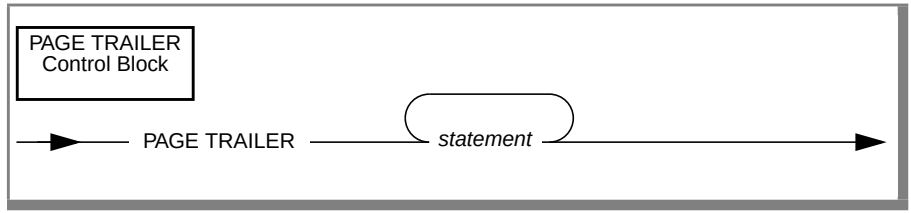
- You can use a PAGE HEADER control block to display column headings in a report.
- You can use the PAGENO expression in a PRINT statement within a PAGE HEADER control block to display the page number automatically at the top of every page.
- ACE delays the processing of the PAGE HEADER control block until the first PRINT, SKIP, or NEED statement to guarantee that any group columns printed in the PAGE HEADER control block have the same values as the columns printed in the ON EVERY ROW control block.

The following example is from the **clist1.ace** report:

```
page header
    print "NUMBER",
      column 9, "NAME",
      column 32, "LOCATION",
      column 54, "ZIP",
      column 62, "PHONE"
    skip 1 line
```

PAGE TRAILER

The PAGE TRAILER control block specifies what information will appear at the bottom of each page of the report.



PAGE TRAILER	are required keywords.
statement	is a list of one or more statements or a compound statement.

Usage

- The vertical size of the page trailer is equal to the number of lines that you specify in the PAGE TRAILER control block. The BOTTOM MARGIN (in the OUTPUT section) affects how close to the bottom of the page ACE displays the page trailer.
- You cannot use the SKIP TO TOP OF PAGE statement in a PAGE TRAILER control block.
- If you use an IF THEN ELSE statement in a PAGE TRAILER control block, the number of lines displayed by the PRINT and SKIP statements following the THEN keyword must be equal to the number of lines displayed by the PRINT and SKIP statements following the ELSE keyword.
- If you use a FOR or WHILE statement that contains a PRINT statement in a PAGE TRAILER control block, you must terminate the PRINT statement with a semicolon. The semicolon suppresses any NEWLINE (RETURN) characters in the loop, keeping the number of lines in the trailer constant from page to page.

- You cannot use the PRINT FILE statement to read and display text from a file in a PAGE TRAILER control block.
- You can use the PAGENO expression in a PRINT statement within a PAGE TRAILER control block to display the page number automatically at the bottom of every page.

The following example is from the **ord3.ace** report:

```
page trailer
   print column 28, pageno using "page <<<<"
```

Statements

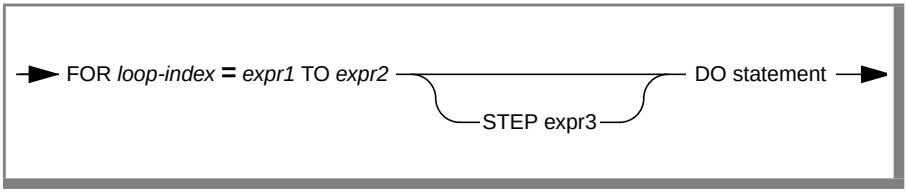
The format control blocks determine *when* ACE takes an action, while the statements determine *what* action ACE takes.

Statements are composed of keywords and expressions, as explained under each of the specific statements.

Any statement can be a single statement or a compound statement. A compound statement is one or more statements, including other compound statements, preceded by the BEGIN keyword and followed by an END keyword.

FOR

The FOR statement defines a loop. It repeatedly executes a simple or compound statement, incrementing the loop index before each pass through the loop. Control passes to the first statement following the end of the loop when the termination condition is satisfied.



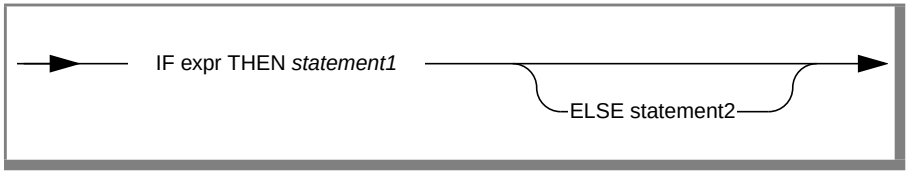
FOR	is a required keyword.
loop-index	is the name of a variable declared in the DEFINE section. The FOR statement uses this variable as the loop index, changing its value each time through the loop.
=	is a required keyword.
expr1	is a required expression that specifies the starting value of the loop index.
TO	is a required keyword.
expr2	is a required expression that specifies the ending value of the loop index. This expression specifies the termination condition for the loop.
STEP	is an optional keyword.
expr3	is an optional expression that specifies the amount the FOR statement increments the loop index each time through the loop. If you do not specify STEP and *expr3*, the FOR statement assumes an increment value of 1.
DO	is a required keyword.
statement	is a single statement or a compound statement.

Usage

- You cannot have a decrementing loop—the value of *expr3* must be positive.
- If a compound statement follows the DO keyword, you must precede the compound statement with a BEGIN keyword and follow it with END.

IF THEN ELSE

This statement defines a conditional branch. It evaluates an expression and executes specific statements based on the result of the evaluation.



IF	is a required keyword.
expr	is a required expression that determines which, if any, of the statements IF executes.
THEN	is a required keyword.
statement1	is a required single statement or compound statement that IF executes if *expr* evaluates as true (not equal to zero).
ELSE	is an optional keyword.
statement2	is an optional single statement or compound statement that IF executes if *expr* evaluates as false (equal to zero).

Usage

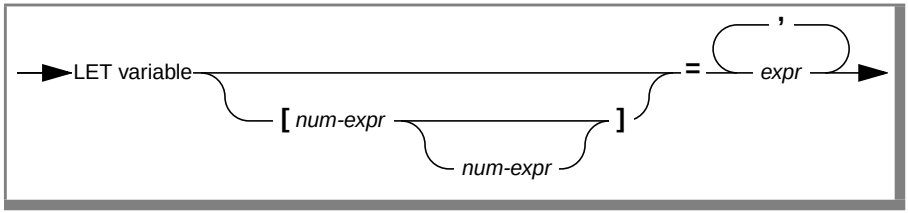
- If a compound statement follows the THEN or ELSE keyword, you must precede the compound statement with the BEGIN keyword and follow it with END.
- You can nest IF THEN ELSE statements to 128 levels.

The following example is from the **mail3.ace** report:

```
if i = count1 then
   begin

      print array1 clipped
      print array2 clipped
      print array3 clipped

      skip 1 line

      let array1 = " "
      let array2 = " "
      let array3 = " "

      let i  = 1
   end

else
   let i = i + 1
```

LET

The LET statement assigns a value to a declared variable.



LET	is a required keyword.
variable	is a required variable name; that is, a variable declared in a previous DEFINE section.
num-expr	is an optional number expression or list of one or two number expressions. They specify a substring of the CHAR variable to which LET is to assign a value. If one *num-expr* is specified, the substring is the beginning character within the variable through the end of the string. If two *num-expr* are present, the substring is from the first *num-expr* through the second one. You can only use these substring operations with a CHAR variable. You must enclose subscripts in brackets ([ ]).
=	is a required keyword.
expr	is a required list of one or more expressions separated by commas. LET assigns the value of this list to the *variable*. If the list contains more than one expression, the *variable* must be of type CHAR. LET assigns the value of the string generated by the concatenation of all expressions in the expression list to the variable. The result that ACE assigns to the variable takes the same form as if you had displayed the same expression list with a PRINT statement. LET accepts all the expressions that PRINT accepts including USING, CLIPPED, ASCII, COLUMN, and subscripted character expressions.

Usage

- If you assign a value with a fractional part to an INTEGER or SMALLINT variable, ACE truncates the fractional part.
- Refer to the descriptions of expressions beginning on page 4-82 for more information about type conversion.

The following example is from the FORMAT section of the **mail3.ace** report:

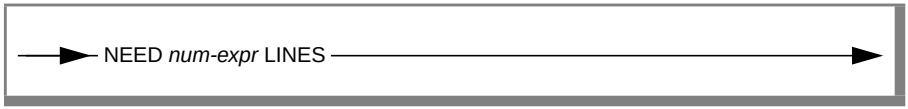
```
let i = 1

let l_size = 72/count1

let white = 8/count1
```

GLS

Conversion of a monetary or numeric value to a character string using the LET statement results in a string containing locale-specific formatting if you enable certain GLS settings. This is true for both the default conversion and the conversion with a USING clause. Refer to Appendix C, "Global Language Support," and the *Informix Guide to GLS Functionality*.

NEED

This statement causes subsequent display to start on the next page if the specified number of lines cannot be placed on the current page.



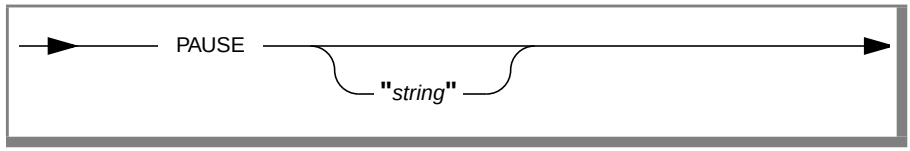
NEED	is a required keyword.
num-expr	is an expression that evaluates to a number specifying the number of lines needed.
LINES	is a required keyword.

Usage

Use the NEED statement to prevent ACE from separating parts of the report that you want to keep together on a single page.

PAUSE

This statement causes output to the terminal to pause until you press RETURN.



PAUSE	is a required keyword.
string	is an optional message that PAUSE displays. If you do not supply a message, PAUSE does not display a message.

Usage

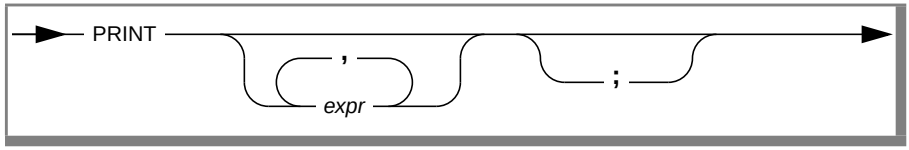
The PAUSE statement has no effect if you use a REPORT TO *filename* or a REPORT TO PRINTER statement in the OUTPUT section.

The following example causes ACE to pause while running the report:

```
after group of item_num
     .
     .
     .
skip to top of page
pause "Press RETURN to continue"
```

PRINT

This statement displays information on the screen or as specified in the OUTPUT section.



PRINT	is a required keyword.
expr	is an optional list of one or more expressions, separated by commas.
;	is an optional keyword that suppresses a NEWLINE (RETURN) at the end of the line.

Usage

- Unless you use the optional WORDWRAP keyword, one PRINT statement displays its output on one line, no matter how many lines the statement occupies in the report specification.
- Unless you use the keyword CLIPPED or USING following an expression, ACE displays an expression so that it occupies a predetermined number of spaces.
- You can name a TEXT column in a PRINT statement. The PRINT statement acts like a PRINT FILE statement with the TEXT item as a file.

Figure 4-3
Default Display Widths

Column	Default Size
CHAR	declared size
DATE	10
FLOAT	14 (including sign and decimal point)
SMALLINT	6 (including sign)

(1 of 2)

Column	Default Size
INTEGER	11 (including sign)
SMALLFLOAT	14 (including sign and decimal point)
DECIMAL	number of digits plus 2 (including sign and decimal point)
SERIAL	11
MONEY	number of digits plus 3 (including sign, decimal point, and dollar sign)
DATETIME	depends on precision
INTERVAL	depends on precision

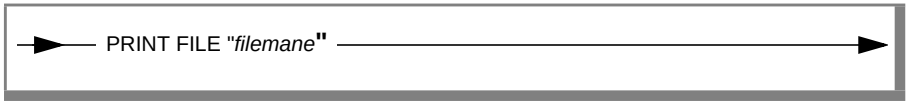
(2 of 2)

The following example is from the **clist2.ace** report:

```
first page header
print column 32, "CUSTOMER LIST"
print column 32, "-------------"
skip 2 lines
print "Listings for the State of ", thisstate
skip 2 lines
print "NUMBER",
    column 9, "NAME",
    column 32, "LOCATION",
    column 54, "ZIP",
    column 62, "PHONE"
skip 1 line

page header
print "NUMBER",
    column 9, "NAME",
    column 32, "LOCATION",
    column 54, "ZIP",
    column 62, "PHONE"
skip 1 line

on every row
print customer_num using "####",
     column 9, fname clipped, 1 space, lname clipped,
     column 32, city clipped, ", " , state,
     column 54, zipcode,
     column 62, phone
```

PRINT FILE

This statement displays the contents of a text file in a report.



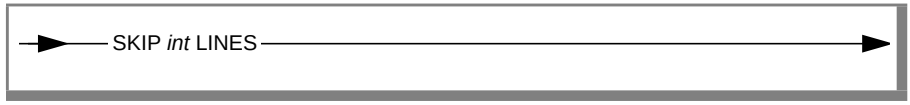
PRINT FILE	are required keywords.
filename	is a required filename that can be a pathname. You must enclose the filename in quotation marks.

Usage

You can use the PRINT FILE statement to include the body of a form letter in a report that generates custom letters.

SKIP

This statement skips lines in a report.



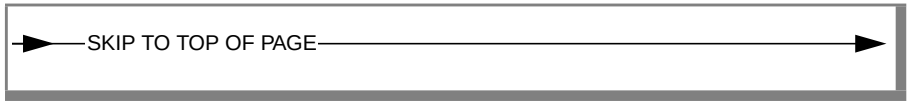
SKIP	is a required keyword.
in	is an integer specifying the number of lines to skip.
LINES	is a required keyword. You can use the keyword LINE in place of LINES if you like.

The following example is from the **mail1.ace** report:

```
format
   on every row
     print fname, lname
     print company
     print address1
     print address2
     print city, ", " , state,
    2 spaces, zipcode
     skip 2 lines
```

SKIP TO TOP OF PAGE

This statement causes subsequent printing to begin at the top of the next page.



Usage

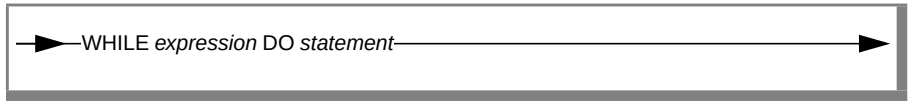
You cannot use a SKIP TO TOP OF PAGE statement in a FIRST PAGE HEADER, PAGE HEADER, or PAGE TRAILER control block.

The following example is from the **mail2.ace** report:

```
format
on every row
   if (city is not null) and
     (state is not null) then
   begin
     print fname clipped, 1 space, lname
     print company
     print address1
     if (address2 is not null) then
        print address2
     print city clipped, ", " , state,
        2 spaces, zipcode
     skip to top of page
   end
end
```

WHILE

The WHILE statement defines a loop that repeatedly executes a simple or compound statement while the expression is true. Control passes to the first statement following the loop when the expression evaluates as false.



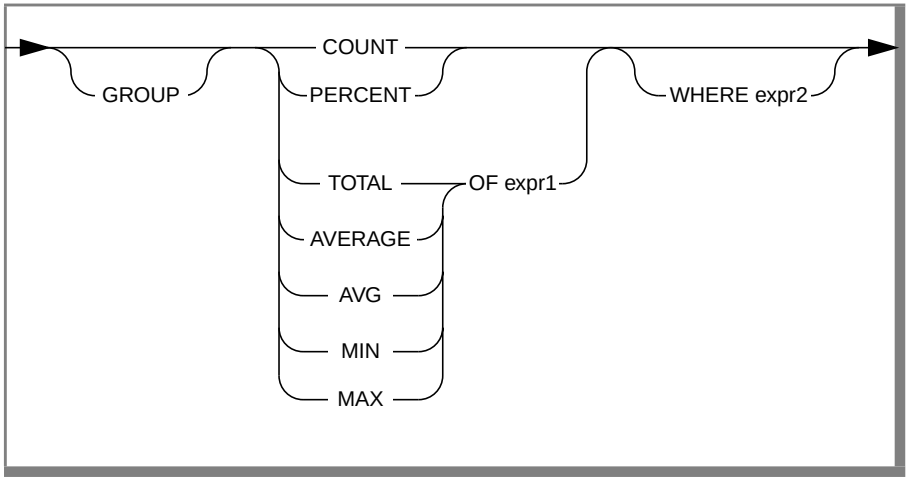
WHILE	is a required keyword.
expression	is a required expression. While this expression evaluates as true, WHILE executes the loop. When this expression evaluates as false, control passes to the first statement following the loop.
DO	is a required keyword.
statement	is a single statement or a compound statement.

Usage

If more than one statement follows the DO keyword, you must precede them with the BEGIN keyword and follow them with END to create a compound statement.

Aggregates

Aggregates allow you to summarize information in a report.



GROUP	is an optional keyword that causes the aggregate to reflect information for a specific group only. The group must have been designated by an ORDER BY clause in the SELECT or READ section of the report specification. You can only use this keyword in an AFTER GROUP OF control block.
COUNT	is a keyword that is evaluated as the total number of rows qualified by the SELECT section or retrieved by the READ section, and qualified by the optional WHERE *expr2* expression that can appear in an aggregate statement.
PERCENT	is the keyword that evaluates COUNT as a percent of the total number of rows qualified by the SELECT section or retrieved by the READ section, and qualified by the optional WHERE *expr2* expression that can appear in an aggregate statement.
TOTAL	evaluates as the total of *expr1* in the rows qualified by the SELECT section or retrieved by the READ section, and qualified by the optional WHERE *expr2* expression that can appear in an aggregate statement.
AVERAGE / AVG	evaluates as the average of *expr1* in the rows qualified by the SELECT section or retrieved by the READ section, and qualified by the optional WHERE *expr2* expression that can appear in an aggregate statement.

MIN	evaluates as the minimum of *expr1* in the rows qualified by the SELECT section or retrieved by the READ section, and qualified by the optional WHERE *expr2* expression that can appear in an aggregate statement.
MAX	evaluates as the maximum of *expr1* in the rows qualified by the SELECT section or retrieved by the READ section, and qualified by the optional WHERE *expr2* expression that can appear in an aggregate statement.
OF	is a keyword.
EXPR1	is the expression that TOTAL, AVERAGE, MIN, and MAX evaluate. It is typically a column or a number expression that includes a number column.
WHERE	is a keyword.
EXPR2	is a logical expression that qualifies the aggregate.

Usage

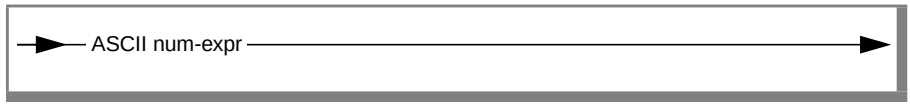
- The WHERE part of an aggregate statement further qualifies rows that the SELECT section already qualified or that the READ statement already retrieved. WHERE cannot select rows that were not qualified by the SELECT section or retrieved by the READ section.
- Aggregates produce unpredictable results when *expr1* or *expr2* contains user-defined variables. (See "PARAM" on page 4-20 and "VARIABLE" on page 4-21.)

The following example is from the **ord2.ace** report:

```
on every row
   print snum using "###", column 10, manu_code,
      column 18, description clipped, column 38,
      quantity using "###", column 43, unit_price
      using "$$$$.&&", column 55,
      total_price using "$$,$$$,$$$.&&"

after group of number

   skip 1 line

   print 4 spaces,
      "Shipping charges for the order:  ",
      ship_charge using "$$$$.&&"

   skip 1 line

   print 5 spaces, "Total amount for the order:  ",
      ship_charge + group total of
      total_price using "$$,$$$,$$$.&&"
```

ASCII

ACE evaluates this expression as a value that you can use as a character. You can use it to display control characters.



ASCII	is a required keyword.
num-expr	is a number expression.

Usage

Do not confuse this keyword with the ASCII keyword used in the DEFINE section to specify the identifiers and data values of an input file.

The following PRINT statement rings the bell (ASCII value of 7) of your computer:

```
print ascii 7
```

The next report specification segments show how to implement special printer or computer functions. They assume that when the printer receives the sequence of ASCII characters 9, 11, and 1, it starts printing in red, and when it receives 9, 11, and 0, it reverts to black printing. The values used in the example are hypothetical; refer to your printer or terminal manual for information on your printer or terminal.

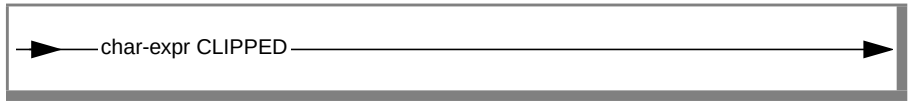
This specification uses the FIRST PAGE HEADER control block to initialize variables that are used in other control blocks.

```
    .
    .
define
     variable red_on  char(3)
     variable red_off char(3)
end
    .
    .
format
   first page header
      let red_on  =
          ascii 9, ascii 11, ascii 1
      let red_off =
          ascii 9, ascii 11, ascii 0

   on every row
      .
      .
      print red_on,
          "Your bill is overdue.",
      red_off
        .
        .
```

Tip: *ACE cannot distinguish printable and nonprintable ASCII characters. Be sure to account for the nonprinting characters when you use the COLUMN expression to format your page. Because various devices print spaces with control characters differently, you might have to use trial and error to line up columns when you print control characters.*

CLIPPED

This expression displays the character field that precedes it without any trailing blanks.



char-expr	is a required character expression.
CLIPPED	is a required keyword.

Usage

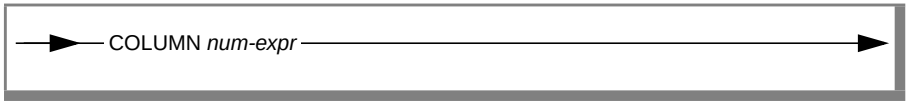
You normally use CLIPPED following a *column-name* in a PRINT statement.

The following example is from the **mail2.ace** report:

```
format
   on every row
      if (city is not null) and
         (state is not null) then
      begin
         print fname clipped, 1 space, lname
         print company
         print address1
         if (address2 is not null) then
            print address2
         print city clipped, ", " , state,
            2 spaces, zipcode
         skip to top of page
      end
end
```

COLUMN

This expression evaluates to a string of spaces long enough to position the next item in the designated column.



COLUMN	is a required keyword.
num-expr	is a required number expression that specifies the column position for the next item to be printed.

Usage

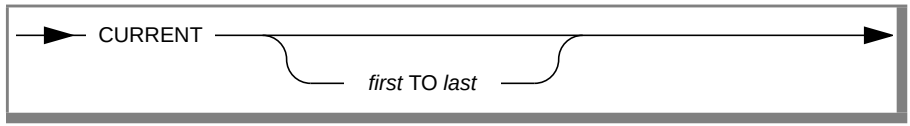
- ACE calculates the column number by adding the number to the left margin you set in the OUTPUT section.
- If ACE has already printed past the column specified by *num-expr*, ACE ignores the COLUMN expression.

The following example is from the **clist2.ace** report:

```
page header
print "NUMBER",
    column 9, "NAME",
    column 32, "LOCATION",
    column 54, "ZIP",
    column 62, "PHONE"
skip 1 line
```

CURRENT

This expression evaluates as a character string with the value of the current date and time as supplied by the operating system.



CURRENT	is a required keyword.
first	is an optional qualifier that specifies the first field to be returned.
TO	is a required keyword if you include *first* and *last* qualifiers.
last	is an optional qualifier that specifies the last field to be returned.

The following example prints the current date and time to a precision of MINUTE:

```
print current year to minute
```

DATE

This expression evaluates as a character string with a value of today's date in the form "Thu Feb 17 1999."

Because DATE evaluates as type CHAR, you can use it with subscripts to express a day, month, date, or year. The following example:

```
print "Today is ", date[1,3]
```

displays the three-letter abbreviation for the day of the week.

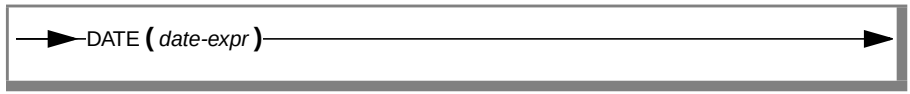
```
Today is Mon
```

GLS

The installation of message files in a subdirectory of **$INFORMIXDIR/msg** and subsequent reference to that subdirectory by way of the environment variable **DBLANG** causes month and day portions of the character string returned by DATE to contain language-specific month name and day name abbreviations. For example, in a Spanish locale the day Saturday is translated into the day name abbreviation *Sab*, which stands for *Sabado* (the Spanish word for Saturday).♦

DATE()

The DATE function converts the expression with which you call it to type DATE.



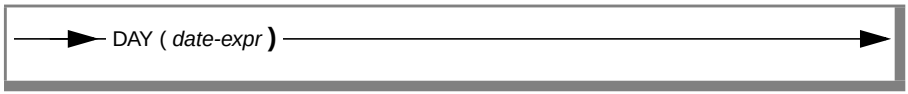
DATE	is a required keyword.
date-expr	is a required expression of any type that evaluates to a type DATE value.

Usage

- The DATE function is typically used to convert date strings to type DATE.
- A properly formatted date string is required. The default format is *mm/dd/yy*, but this can be changed by way of the environment variable **DBDATE**.

DAY( )

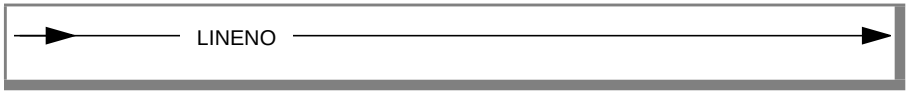
The DAY function returns the day of the month when you call it with a type DATE or DATETIME expression.



DAY	is a required keyword.
date-expr	is a required expression that evaluates to a type DATE or DATETIME value.

LINENO

This expression has the value of the line number of the line that ACE is currently displaying. ACE computes the line number by calculating the number of lines from the top of the page.

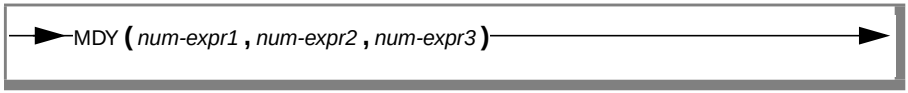


Usage

Do not use LINENO within a page header or trailer. LINENO works on the first page header but does not work on any subsequent pages.

MDY( )

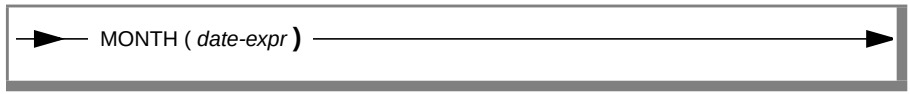
The MDY function returns a type DATE value when you call it with three expressions that evaluate to integers representing the month, date, and year.



MDY	is a required keyword.
num-expr1	is an expression that evaluates to an integer that represents the number of the month (1-12).
num-expr2	is an expression that evaluates to an integer that represents the number of the day of the month (1-28, 29, 30, or 31, depending on the month).
num-expr3	is an expression that evaluates to a four-digit integer that represents the year.

MONTH( )

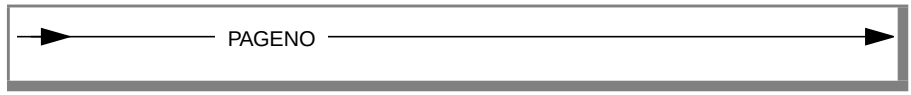
The MONTH function returns an integer that corresponds to the month (1-12) of its type DATE or DATETIME argument.



MONTH	is a required keyword.
date-expr	is a required expression that evaluates to a type DATE or DATETIME value.

PAGENO

This expression has the value of the page number of the page that ACE is currently displaying.



Usage

Use PAGENO in a PRINT statement in the PAGE HEADER or PAGE TRAILER control block to number the pages of a report. (You can also use PAGENO in other control blocks.)

The following example is from the **ord3.ace** report:

```
page trailer
   print column 28, pageno using "page <<<<"
```

SPACES

This expression evaluates as a string of spaces. It is identical to a quoted string of spaces.

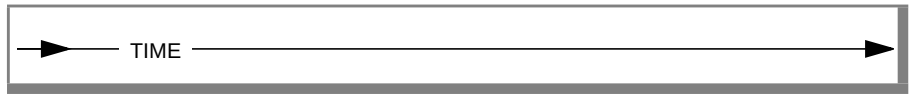
num-expr	is a number expression that designates how many spaces.
SPACE[S]	is a required keyword. You can use either the keyword SPACE or SPACES.

The following example is from the **mail1.ace** report:

```
format
  on every row
     print fname, lname
     print company
     print address1
     print address2
     print city, ", " , state,
    2 spaces, zipcode
     skip 2 lines
  end
```

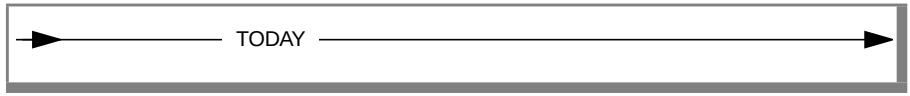
TIME

This expression evaluates as a character string with a value of the current time in the form *hh:mm:ss*.



TODAY

This expression evaluates as type DATE with a value of the current date as supplied by the operating system.

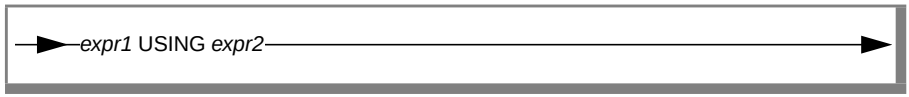


The following example is from the **ord3.ace** report:

```
skip 1 line
  print column 15, "FROM:  ", begin_date
     using "mm/dd/yy",
  column 35, "TO:  ", end_date
     using "mm/dd/yy"
  print column 15, "Report run date:  ",
     today using "mmm dd, yyyy"
  skip 2 lines
  print column 2, "ORDER DATE", column 15,
     "COMPANY", column 35, "NAME",
     column 57, "NUMBER", column 65, "AMOUNT"
```

USING

This expression allows you to format a number or date expression. With a number expression, you can use USING to line up decimal points, right- or left-justify numbers, put negative numbers in parentheses, and perform other formatting functions. With a date expression, USING converts the date to a variety of formats.



expr1	is the required expression that specifies what USING is to format.
USING	is a required keyword.
expr2	is the required format string that specifies how USING is to format *expr1*.

Usage

- The format string must appear between quotation marks.
- Although USING is generally used as part of a PRINT statement, you can also use it with LET.
- If you attempt to display a number that is too large for a display field, ACE fills the field with asterisks to indicate an overflow.

Formatting Number Expressions

The format string consists of combinations of the following characters: * & # < , . - + ( ) $. The following characters *float*: - + ( ) $. When a character floats, ACE displays multiple leading occurrences of the character as a single character as far to the right as possible, without interfering with the number that is being displayed. Refer to the following list:

* This character fills with asterisks any positions in the display field that would otherwise be blank.

& This character fills with zeros positions in the display field that would otherwise be blank.

\# This character does not change any blank positions in the display field. Use this character to specify a maximum width for a field.

< This character causes the numbers in the display field to be left-justified.

, This character is a placeholder for the thousands separator. The default thousands separator is a comma. Environment variables DBFORMAT, DBMONEY, and LANG determine the thousands separator.

. This character is a placeholder for the decimal separator. The default decimal separator is a period. Environment variables DBFORMAT, DBMONEY, and LANG determine the decimal separator.

\- This character is a literal; USING displays it as a minus sign when *expr1* is less than zero. When you group several in a row, a single minus sign floats to the rightmost position without interfering with the number being printed.

\+ This character is a literal; USING displays it as a plus sign when *expr1* is greater than or equal to zero and as a minus sign when it is less than zero. When you group several in a row, a single plus sign floats to the rightmost position without interfering with the number being printed.

(This character is a literal; USING displays it as a left parenthesis before a negative number. It is the accounting parenthesis that is used in place of a minus sign to indicate a negative number. When you group several in a row, a single left parenthesis floats to the rightmost position without interfering with the number being printed.

) This is the accounting parenthesis that is used in place of a minus sign to indicate a negative number. A single one of these characters generally closes a format string that begins with a left parenthesis.

$ This character is a placeholder for the leading currency symbol. The default leading currency symbol is a dollar sign. Environment variables DBFORMAT, DBMONEY, and LANG determine the leading currency symbol. When you group several in a row, a single dollar sign (or leading specified currency symbol) floats to the rightmost position without interfering with the number being printed.

@ This character is a placeholder for the trailing currency symbol.The default trailing currency symbol defaults to NULL. Environment variables DBFORMAT, DBMONEY, and LANG determine the trailing currency symbol. When you group several in a row, a single at sign (or locale specified trailing currency symbol) floats to the leftmost position without interfering with the number being printed.

Refer to "Sample Format Strings" on page 4-101 for examples of formatting number expressions.

Formatting Date Expressions

The format string consists of combinations of the characters *m*, *d*, and *y* that Figure 4-4 shows.

Figure 4-4
Combinations of Date Format Characters

Format Substring	Formatted Result
dd	Day of the month as a two-digit number (01-31)
ddd	Day of the week as a three-letter abbreviation (Sun through Sat)
mm	Month as a two-digit number (01-12)
mmm	Month as a three-letter abbreviation (Jan through Dec)
yy	Year as a two-digit number in the 1900s (00-99)
yyyy	Year as a four-digit number (0001-9999)

Figure 4-5 shows some sample conversions for December 25, 1999.

Figure 4-5
Results of Date Format Strings

Format String	Formatted Result
"mmddyy"	122599
"ddmmyy"	251299
"yymmdd"	991225
"yy/mm/dd"	99/12/25
"yy mm dd"	99 12 25
"yy-mm-dd	99-12-25
"mmm. dd, yyyy"	Dec. 25, 1999
"mmm dd yyyy"	Dec 25 1999
"yyyy dd mm"	1999 25 12
"mmm dd yyyy"	Dec 25 1999
"ddd, mmm. dd, yyyy"	Tue, Dec. 25, 1999
"(ddd) mmm. dd, yyyy"	(Tue) Dec. 25, 1999

GLS

GLS settings can affect the way the format string in the USING expression is interpreted for numeric and monetary data. In the format string, the period symbol (.) is not a literal character but a placeholder for the decimal separator specified by environment variables. Likewise, the comma symbol (,) is a placeholder for the thousands separator specified by environment variables. The $ symbol is a placeholder for the leading currency symbol. The @ symbol is a placeholder for the trailing currency symbol. Thus, the format string *$$#,###.##* will format the value 1234.56 as $1,234.56 in a US English locale but as DM1.234,56 in a German locale. Note that setting the DBFORMAT or DBMONEY environment variables override settings in LC variables. Refer to Appendix C, "Global Language Support," and the *Informix Guide to GLS Functionality*.

The installation of locale files in a subdirectory of **$INFORMIXDIR/msg** and subsequent reference to that subdirectory by way of the environment variable **DBLANG** causes *mmm* and *ddd* specifiers in a format string to display locale-specific month name and day name abbreviations on the form. For example, the *ddd.* specifier in a Spanish locale translates the day Saturday into the day name abbreviation *Sab.*, which stands for *Sabado* (the Spanish word for Saturday). ♦

The following example prints the balance field using a format string that allows up to $9,999,999.99 to be formatted correctly:

```
print "The current balance is ",
   23485.23 using "$#,###,##&.&&"
```

The result of executing this PRINT statement with the value 23,485.23 follows:

```
The current balance is $   23,485.23
```

This example fixes the dollar sign. If dollar signs had been used instead of # characters, the dollar sign would have floated with the size of the number. It also uses a mix of # and & fill characters. The # character provides blank fill for unused character positions, while the & character provides zero filling. This format ensures that even if the number is zero, the positions marked with & characters appear as zeros, not blanks.

The tables on the following pages illustrate the results of various combinations of data and USING format strings.

Sample Format Strings

Format String	Data Value	Formatted Result
"#####"	0	bbbbb
"&&&&&"	0	00000
"$$$$$"	0	bbbb$
"*****"	0	*****
"<<<<<"	0	(*NULL string*)

Here the character b represents a blank or space.

(1 of 6)

Format String	Data Value	Formatted Result
"<<<,<<<"	12345	12,345
"<<<,<<<"	1234	1,234
"<<<,<<<"	123	123
"<<<,<<<"	12	12
"##,###"	12345	12,345
"##,###"	1234	b1,234
"##,###"	123	bbb123
"##,###"	12	bbbb12
"##,###"	1	bbbbb1
"##,###"	-1	bbbbb1
"##,###"	0	bbbbbb
"&&,&&&"	12345	12,345
"&&,&&&"	1234	01,234
"&&,&&&"	123	000123
"&&,&&&"	12	000012
"&&,&&&"	1	000001
"&&,&&&"	-1	000001
"&&,&&&"	0	000000
"&&,&&&.&&"	12345.67	12,345.67
"&&,&&&.&&"	1234.56	01,234.56
"&&,&&&.&&"	123.45	000123.45
"&&,&&&.&&"	0.01	000000.01
"$$,$$$"	12345	****** (*overflow*)

Here the character b represents a blank or space.

(2 of 6)

Format String	Data Value	Formatted Result
"$$,$$$"	1234	$1,234
"$$,$$$"	123	bb$123
"$$,$$$"	12	bbb$12
"$$,$$$"	1	bbbb$1
"$$,$$$"	0	bbbbb$
"**,***"	12345	12,345
"**,***"	1234	*1,234
"**,***"	123	***123
"**,***"	12	****12
"**,***"	1	*****1
"**,***"	0	******
"##,###.##"	12345.67	12,345.67
"##,###.##"	1234.56	b1,234.56
"##,###.##"	123.45	bbb123.45
"##,###.##"	12.34	bbbb12.34
"##,###.##"	1.23	bbbbb1.23
"##,###.##"	0.12	bbbbb0.12
"##,###.##"	0.01	bbbbbb.01
"##,###.##"	-0.01	bbbbbb.01
"##,###.##"	-1	bbbbb1.00
"$$,$$$.$$"	12345.67	********* (*overflow*)
"$$,$$$.$$"	1234.56	$1,234.56
"$$,$$$.##"	0.00	$.00

Here the character b represents a blank or space.

(3 of 6)

Format String	Data Value	Formatted Result
"$$,$$$.##"	1234.00	$1,234.00
"$$,$$$.&&"	0.00	$.00
"$$,$$$.&&"	1234.00	$1,234.00
"-$$$,$$$.&&"	-12345.67	-$12,345.67
"-$$$,$$$.&&"	-1234.56	-b$1,234.56
"-$$$,$$$.&&"	-123.45	-bbb$123.45
"--$$,$$$.&&"	-12345.67	-$12,345.67
"--$$,$$$.&&"	-1234.56	-$1,234.56
"--$$,$$$.&&"	-123.45	-bb$123.45
"--$$,$$$.&&"	-12.34	-bbb$12.34
"--$$,$$$.&&"	-1.23	-bbbb$1.23
"-##,###.##"	-12345.67	-12,345.67
"-##,###.##"	-123.45	-bbb123.45
"-##,###.##"	-12.34	-bbbb12.34
"--#,###.##"	-12.34	-bbb12.34
"---,###.##"	-12.34	-bb12.34
"---,-##.##"	-12.34	-12.34
"---,--#.##"	-1.00	-1.00
"-##,###.##"	12345.67	12,345.67
"-##,###.##"	1234.56	1,234.56
"-##,###.##"	123.45	123.45
"-##,###.##"	12.34	12.34
"--#,###.##"	12.34	12.34

Here the character b represents a blank or space.

(4 of 6)

Format String	Data Value	Formatted Result
"---,###.##"	12.34	12.34
"---,-##.##"	12.34	12.34
"---,---.##"	1.00	1.00
"---,---.--"	-.01	-.01
"---,---.&&"	-.01	-.01
"----,--$.&&"	-12345.67	-$12,345.67
"----,--$.&&"	-1234.56	-$1,234.56
"----,--$.&&"	-123.45	-$123.45
"----,--$.&&"	-12.34	-$12.34
"----,--$.&&"	-1.23	-$1.23
"----,--$.&&"	-.12	-$.12
"$***,***.&&"	12345.67	$*12,345.67
"$***,***.&&"	1234.56	$**1,234.56
"$***,***.&&"	123.45	$****123.45
"$***,***.&&"	12.34	$*****12.34
"$***,***.&&"	1.23	$******1.23
"$***,***.&&"	.12	$*******.12
"($$$,$$$.&&)"	-12345.67	($12,345.67)
"($$$,$$$.&&)"	-1234.56	(b$1,234.56)
"($$$,$$$.&&)"	-123.45	(bbb$123.45)
"(($$,$$$.&&)"	-12345.67	($12,345.67)
"(($$,$$$.&&)"	-1234.56	($1,234.56)
"(($$,$$$.&&)"	-123.45	(bb$123.45)

Here the character b represents a blank or space.

(5 of 6)

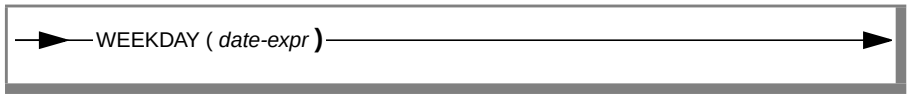
Format String	Data Value	Formatted Result
"(($$,$$$.&&)"	-12.34	(bbb$12.34)
"(($$,$$$.&&)"	-1.23	(bbbb$1.23)
"((((,(($.&&)"	-12345.67	($12,345.67)
"((((,(($.&&)"	-1234.56	($1,234.56)
"((((,(($.&&)"	-123.45	($123.45)
"((((,(($.&&)"	-12.34	($12.34)
"((((,(($.&&)"	-1.23	($1.23)
"((((,(($.&&)"	-.12	($.12)
"($$$,$$$.&&)"	12345.67	$12,345.67
"($$$,$$$.&&)"	1234.56	$1,234.56
"($$$,$$$.&&)"	123.45	$123.45
"(($$,$$$.&&)"	12345.67	$12,345.67
"(($$,$$$.&&)"	1234.56	$1,234.56
"(($$,$$$.&&)"	123.45	$123.45
"(($$,$$$.&&)"	12.34	$12.34
"(($$,$$$.&&)"	1.23	$1.23
"((((,(($.&&)"	12345.67	$12,345.67
"((((,(($.&&)"	1234.56	$1,234.56
"((((,(($.&&)"	123.45	$123.45
"((((,(($.&&)"	12.34	$12.34
"((((,(($.&&)"	1.23	$1.23
"((((,(($.&&)"	.12	$.12

Here the character b represents a blank or space.

(6 of 6)

WEEKDAY( )

The WEEKDAY function returns an integer that represents the day of the week when you call it with a type DATE or DATETIME expression.

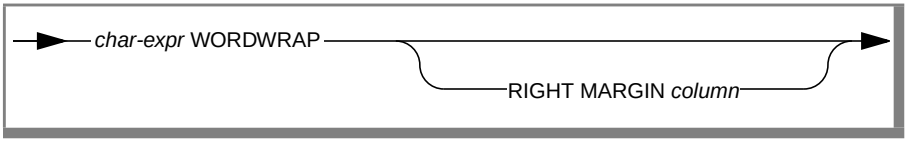


WEEKDAY	is a required keyword.
date-expr	is a required expression that evaluates to a type DATE or DATETIME value.

WEEKDAY returns an integer in the range 0-6. 0 represents Sunday, 1 represents Monday, and so forth.

WORDWRAP

This expression displays the character field that precedes it on multiple lines with lines broken between words at temporary left and right margins.



char-expr	is an expression with a CHAR value.
WORDWRAP	is a required keyword.
RIGHT MARGIN	introduces a temporary right margin that overrides the right margin of the report.
column	specifies the column number of the temporary right margin.

Usage

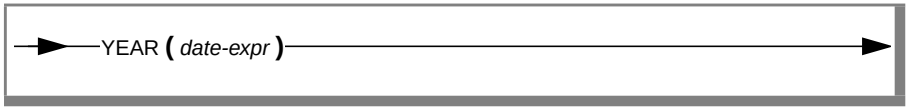
- The temporary left margin is the current printing column. The contents of *char-expr* are displayed on as many lines as necessary between the temporary left and right margins.
- Line breaks are positioned to avoid breaking words where possible.
- A line break is forced where the data contains a line feed (ASCII 10), a return (ASCII 13), or a combination of the two.

The following example invokes WORDWRAP with a temporary right margin at column 70:

```
print column 10, textcol wordwrap right margin 70
```

YEAR( )

The YEAR function returns an integer that represents the year when you call it with a type DATE or DATETIME expression.



YEAR	is a required keyword.
date-expr	is a required expression that evaluates to a type DATE or DATETIME value.

Chapter

5

User-Menu

In This Chapter

The User-menu option allows you to create and run custom menus. The options on a user-menu can call the following items:

- Submenus
- INFORMIX-SQL programs (PERFORM, for example)
- Other programs or sets of programs in your software library
- Operating system utilities
- Forms or reports

Use a special PERFORM screen form to create or alter a menu structure. Two special tables hold the menus, text, and command references for each menu option in the menu structure.

You can create one user-menu for each database. You cannot create a user-menu without specifying a database. The options in a user-menu, however, do not have to refer to any particular database.

A user-menu cannot exist separately from a database. If you want to keep the user-menu apart from your working databases, create a database that contains only menu data.

This chapter contains six sections:

- Accessing a Menu
- Designing a Menu
- Creating a Menu
- Modifying a Menu
- Menu Display Fields
- Creating a Script Menu

Accessing a Menu

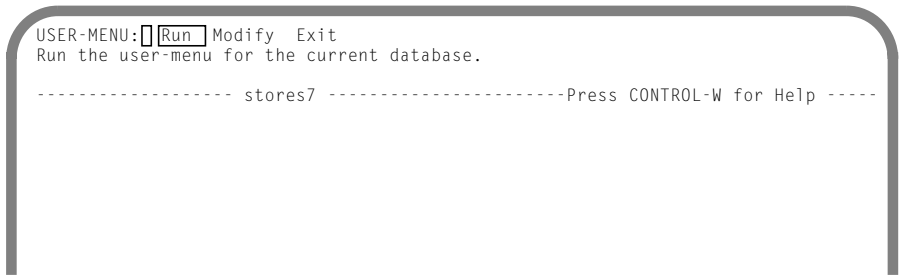
You can access a menu from the INFORMIX-SQL Main menu or from the operating-system command line. The next section describes the use of the User-menu option on the INFORMIX-SQL Main menu. Appendix G explains how to access a user-menu directly from the operating-system command line.

Using a Menu Within INFORMIX-SQL

Follow these steps to access the user-menu included with the **stores7** demonstration database:

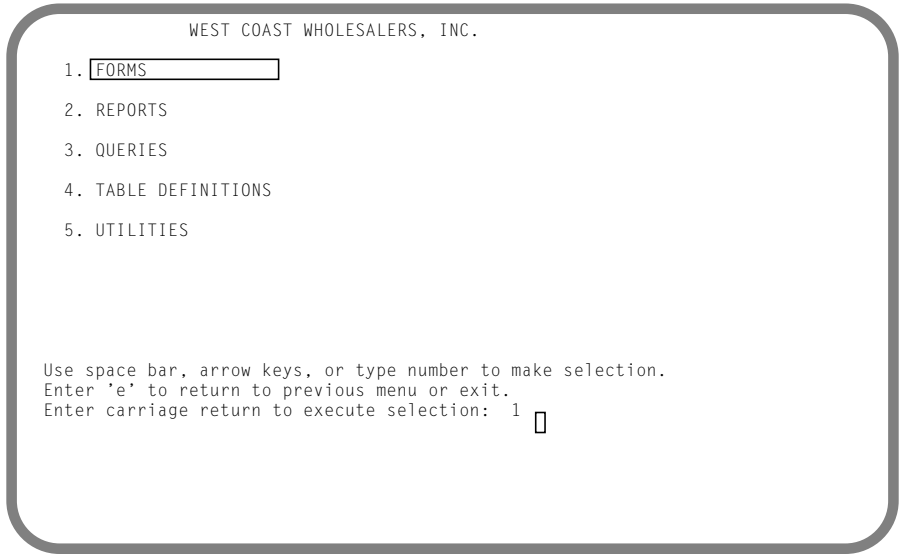
1. Use the Database option on the INFORMIX-SQL Main menu to make the **stores7** database your current database.
2. Select the User-menu option on the INFORMIX-SQL Main menu. The USER-MENU menu displays, as Figure 5-1 shows.

Figure 5-1
The USER-MENU Menu



3. Select the Run option on the USER-MENU menu. The Main menu of the USER-MENU for the **stores7** database displays, as Figure 5-2 on page 5-5 shows.

Figure 5-2
The Main Menu of the stores7 User-Menu



4. Select a menu option by typing the number to the left of the desired option or positioning the highlight on the option with the Arrow keys. Press RETURN. INFORMIX-SQL executes the option you select.

5. Type e to exit a menu. If you type e from the demonstration database menu, the INFORMIX-SQL USER-MENU menu displays.

Designing a Menu

You can have up to 19 levels of menus in the user-menu structure and up to 28 options on each menu.

The total number of options in a single menu depends on two factors:

- The number of lines your screen can hold
- The length of the menu titles

Most screens can accommodate 14 single-spaced menu lines. Each menu line can display 2 options of up to 33 characters of text in each option. If the text for each option on a menu does not exceed 33 characters, you can display up to 14 double-spaced menu options (2 options per line) or 28 single-spaced menu options (2 options per line). If the text for an option is longer than 33 characters, it requires an entire line, reducing the total number of options available for that menu.

As an example of a simple menu, consider the user-menu for the **stores7** demonstration database. It has five options on the top or main level:

- FORMS
- REPORTS
- QUERIES
- TABLE DEFINITIONS
- UTILITIES

Each option on the Main menu calls a menu, and each option on a menu performs an action. A design outline for the user-menu included with the **stores7** database might look like this:

```
WEST COAST WHOLESALERS, INC.

1.   USE DATA ENTRY FORMS
     1.  CUSTOMER ENTRY/QUERY FORM
     2.  ORDER ENTRY/QUERY FORM
     3.  DISPLAY CUSTOMER FORM SPECIFICATION
     4.  DISPLAY ORDER FORM SPECIFICATION

2.   RUN REPORTS
     1.  RUN CUSTOMER REPORT
     2.  RUN REPORT ON CUSTOMER BY DESIGNATED STATE
     3.  RUN CUSTOMER MAILING LABELS
     4.  RUN MATRIX REPORT ON MONTHLY SALES
     5.  DISPLAY REPORT 1 SPECIFICATION
     6.  DISPLAY REPORT 2 SPECIFICATION
     7.  DISPLAY REPORT 3 SPECIFICATION
     8.  DISPLAY REPORT 4 SPECIFICATION

3.   EXECUTE DATABASE QUERIES
     1.  DISPLAY CUSTOMER INFORMATION BASED ON
         PARTIAL NAME MATCH
     2.  INSERT, UPDATE, SELECT, AND DELETE NEW
         CUSTOMER ROW
     3.  DISPLAY ALL CURRENTLY UNPAID ORDERS
     4.  DISPLAY INFORMATION BASED ON A VIEW
     5.  DISPLAY CUSTOMERS PLUS OUTSTANDING ORDERS
         (NO OUTER JOIN)

4.   DISPLAY DATABASE TABLE DEFINITIONS
     1.  TABLE "CUSTOMER"
     2.  TABLE "ITEMS"
     3.  TABLE "MANUFACT"
     4.  TABLE "ORDERS"
     5.  TABLE "STOCK"
     6.  TABLE "STATE"
     7.  TABLE "CUST_CALLS"

5.   UTILITY MENU
     1.  DISPLAY DATE AND TIME
     2.  CHECK CUSTOMER TABLE
     3.  UNLOAD CUSTOMER TABLE
```

Creating a Menu

The first step in creating a menu is to access the PERFORM program with the **menuform** screen form. The second step is to enter data through this screen form.

The **menuform** form is a special PERFORM screen form that you use only to create or to modify a user-menu. You can enter, change, and remove menu information with the form, but you cannot change the appearance of the screen form itself, and you cannot run FORMBUILD on it.

Accessing PERFORM with the menuform Form

Select the User-menu option on the INFORMIX-SQL Main menu. If there is no current database, the CHOOSE DATABASE screen appears. After you select a database, the USER-MENU menu displays. Select the Modify option and INFORMIX-SQL displays the PERFORM menu with the **menuform** form (see Figure 5-3).

Figure 5-3
The PERFORM Menu with the menuform Entry Form

```
PERFORM: Query  Next  Previous  View  Add  Update  Remove  Table  . . .
Searches the active database table.              ** 1: sysmenus table**

==========================MENU ENTRY FORM================================


Menu Name:  [          ]

Menu Title: [                                                          ]

--------------------------SELECTION SECTION--------------------------------

Selection Number:                      Selection Type:

Selection
Text:

Selection
Action:
```

Two tables are accessed in the **menuform** form:

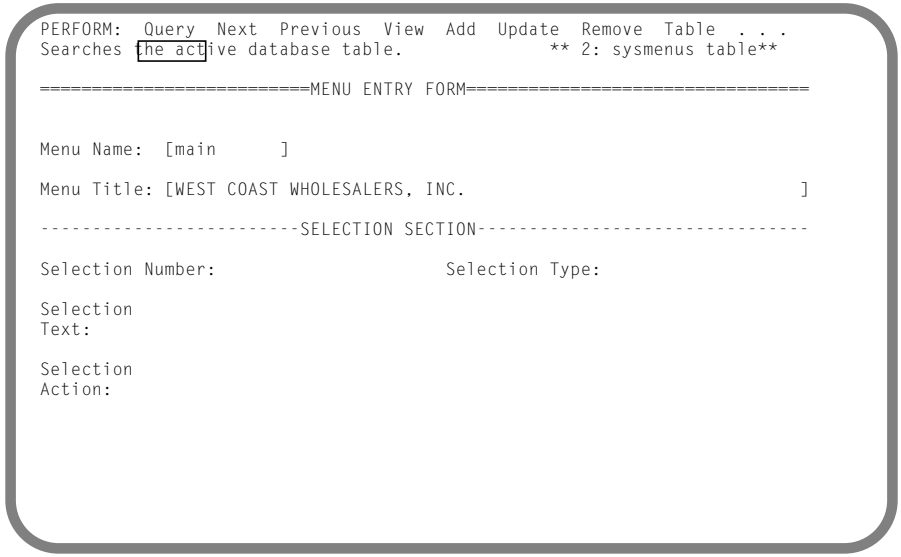
- The **sysmenus** table stores information about each menu in the user-menu. This information includes the Menu name (used by INFORMIX-SQL to identify the menu), and the Menu title (text that you want to appear when the menu displays).
- The **sysmenuitems** table stores information about the options on each menu. This information includes the following:
  - The option number
  - The option type (program, report, form, INFORMIX-SQL command file, script menu, or menu)
  - The option title (text that appears on the screen)
  - The action the option specifies (execute a program, report, form, or script menu, or call an INFORMIX-SQL command file or a menu). A script menu allows the user to run multiple actions in sequence for a single item. Script menus are described in "Creating a Script Menu" on page 5-25.

The **sysmenus** table is the master of the **sysmenuitems** table.

The first person to press Modify from the USER-MENU menu creates both tables. The creator of the **sysmenus** and **sysmenuitems** tables is also the owner of those tables. Ownership is important when an ANSI-compliant database is created because, though anyone can run the user-menu, only the owner of the tables can modify the menu items. Any other user who tries to modify the menu items receives an error message.

Entering Menu Data

You enter menu data in two steps. First you enter data for a menu in the two fields at the top of the screen form (the fields associated with the **sysmenus** table). For example, the entry into the **sysmenus** table for the Main menu in the **stores7** database is shown in this screen.



The following table shows the information for the Main menu and its options, which are stored in the **sysmenus** table in the **stores7** database.

Menu Name	Menu Title
main	WEST COAST WHOLESALERS, INC.
forms	FORMS
reports	REPORTS
queries	DATABASE QUERIES
tables	DATABASE TABLE DEFINITIONS
utilities	UTILITIES

Second, you enter data for each option of each menu in the fields on the lower half of the screen (the fields associated with the **sysmenuitems** table).

You must make an entry in the **sysmenus** table before you make entries for a menu option in the **sysmenuitems** table. This step is necessary because INFORMIX-SQL checks when you enter data in the **sysmenuitems** table to make sure that menus cited in that table exist in the **sysmenus** table.

For example, the complete entry for option 1 on the Main menu is shown in this screen.

```
PERFORM: Query  Next  Previous  View  Add  Update  Remove  Table  . . .
Searches the active database table.              ** 2: sysmenuitems table**

=========================MENU ENTRY FORM================================

Menu Name:  [main              ]

Menu Title:  WEST COAST WHOLESALERS, INC.

-------------------------SELECTION SECTION--------------------------------

Selection Number:  [1      ]            Selection Type:  [M]

Selection
Text:      [FORMS                                                      ]

Selection
Action:    [forms                                                      ]
```

In this instance, the Selection Action (forms) is the name of a menu (corresponding to Selection Type M). Information about the FORMS menu must be entered into the **sysmenus** table before you complete the entry for option 1.

The information stored in the **sysmenuitems** table in the **stores7** database appears in Figure 5-4 on page 5-12.

Figure 5-4
Data in the sysmenuitems Table

Menu Name	Option	Selection Text	Type	Action
main	1	FORMS	M	forms
main	2	REPORTS	M	reports
main	3	QUERIES	M	queries
main	4	TABLE DEFINITIONS	M	tables
main	5	UTILITIES	M	utilities
forms	1	CUSTOMER ENTRY / QUERY FORM	F	customer
forms	2	ORDER ENTRY / QUERY FORM	F	orders
forms	3	DISPLAY CUSTOMER FORM SPECIFICATION	P	type customer.per
forms	4	DISPLAY ORDER FORM SPECIFICATION	P	type orders.per
reports	1	RUN CUSTOMER REPORT	R	clist1
reports	2	RUN REPORT ON CUSTOMER BY DESIGNATED STATE	R	clist2
reports	3	RUN CUSTOMER MAILING LABELS	S	mailinglabels
reports	4	RUN MATRIX REPORT ON MONTHLY SALES	R	months
reports	5	DISPLAY REPORT1 SPECIFICATION	P	type clist1.ace
reports	6	DISPLAY REPORT2 SPECIFICATION	P	type clist2.ace
reports	7	DISPLAY REPORT3 SPECIFICATION	P	type mail.ace
reports	8	DISPLAY REPORT4 SPECIFICATION	P	type months.ace
mailinglabels	1	run mailing labels report	R	mail
mailinglabels	2	display output file from mailing labels report	P	type mail.out
queries	1	DISPLAY CUSTOMER INFORMATION BASED ON PARTIAL NAME MATCH	S	query1
queries	2	INSERT, UPDATE, SELECT, AND DELETE NEW CUSTOMER ROW	S	query2

(1 of 3)

Menu Name	Option	Selection Text	Type	Action
queries	3	DISPLAY ALL CURRENTLY UNPAID ORDERS	S	query3
queries	4	DISPLAY INFORMATION BASED ON A VIEW	S	query4
queries	5	DISPLAY CUSTOMERS PLUS OUTSTANDING ORDERS	S	query5
query1	1	display SQL syntax for query menu choice 1	P	type cust_nme.sql
query1	2	run query menu choice 1	Q	cust_nme
query2	1	display SQL syntax for query menu choice 2	P	type cust_row.sql
query2	2	run query menu choice 2	Q	cust_row
query3	1	display SQL syntax for query menu choice 3	P	type unpaid.sql
query3	2	run query menu choice 3	Q	unpaid
query4	1	display SQL syntax for query menu choice 4	P	type view_c.sql ; type view_s.sql ; type view_d.sql
query4	2	run query menu choice 4	Q	view_s
query5	1	display SQL syntax for query menu choice 5	P	type orders1.sql ; type orders2.sql
query5	2	run query menu choice 5	Q	orders1
query5	3	run query menu choice 5	Q	orders2
tables	1	TABLE “CUSTOMER”	P	type c_custom.sql
tables	2	TABLE “ITEMS”	P	type c_items.sql
tables	3	TABLE “MANUFACT”	P	type c_manuf.sql
tables	4	TABLE “ORDERS”	P	type c_orders.sql
tables	5	TABLE “STOCK”	P	type c_stock.sql
tables	6	TABLE “STATE”	P	type c_state.sql
tables	7	TABLE “CUST_CALLS”	P	type c_custcl.sq1

(2 of 3)

Menu Name	Option	Selection Text	Type	Action
utilities	1	DISPLAY DATE AND TIME	P	date
utilities	2	CHECK CUSTOMER TABLE	Q	ch_cust
utilities	3	UNLOAD CUSTOMER TABLE	S	utility3
utility3	1	display SQL syntax for utility menu choice 3	P	type u_cust.sql
utility3	2	run utility menu choice 3	Q	u_cust
utility3	3	display output of utility menu choice 3	P	type customer.unl

(3 of 3)

Steps for Entering Your Own Data

The following steps describe the procedure for entering your own menu and option data. See the list of display fields in "Menu Display Fields" on page 5-16 for information on the kind of data to enter for each display field.

1. Select the User-menu option on the INFORMIX-SQL Main menu.
2. Select the Modify option on the USER-MENU Menu. The MENUFORM screen form displays.
3. Type `a` to select the Add option.
4. Enter a menu name and menu title for the first menu. Note that `main` must be the first menu name. Press ESCAPE when you finish.
5. Enter data in the same way for the second menu. Press ESCAPE when you finish.
6. When you have entered the menu name and menu title data for all menus in the user-menu, you are ready to enter data into the SELECTION SECTION of the form. You should now enter information about all options on the Main menu. Select the Query option, enter `main` in the Menu Name field, and press ESCAPE.
7. Type `d` to make the detail table (**sysmenuitems**) active. INFORMIX-SQL displays the following message:

   ```
   There are no rows satisfying the conditions.
   ```

   The **sysmenuitems** table contains no rows joining the `main` entry and the Menu Name field.

8. Select the Add option. Enter the Selection Number, Selection Type, Selection Text, and Selection Action data for the first option on the Main Menu. Press ESCAPE when you finish entering data about the first menu option. If there is a second option to the Main Menu, select Add and enter data about that option. Press ESCAPE when you finish.

   Repeat this step until you have entered data for each option on the Main Menu.

9. Type `m` to call the master table again. Use the Query option to locate and display the Menu Name and Menu Title data for your next menu. Type `d` to display the detail table joined to the current row of the master table.

10. Enter the Selection Number, Selection Type, Selection Text, and Selection Action data for the first option on this menu. Press ESCAPE when you finish entering data. Repeat this step for each option in this menu.

11. Repeat steps 9 and 10 to enter data for the remaining menu options. When you have entered data for all the options in each menu, the menu is complete. Select the Exit option to leave PERFORM and return to the USER-MENU menu.

12. Select the Run option on the USER-MENU menu to run the new menu.

Modifying a Menu

You change a user-created menu in the same way you create one. Select the User-menu option from the INFORMIX-SQL Main menu. Then select the Modify option on the USER-MENU menu. Use the PERFORM options to modify the menu entries in the MENUFORM screen form.

For information about PERFORM, see Chapter 3, "The PERFORM Screen Transaction Processor."

Menu Display Fields

This section discusses the kinds of information you can enter for each field in the MENUFORM screen.

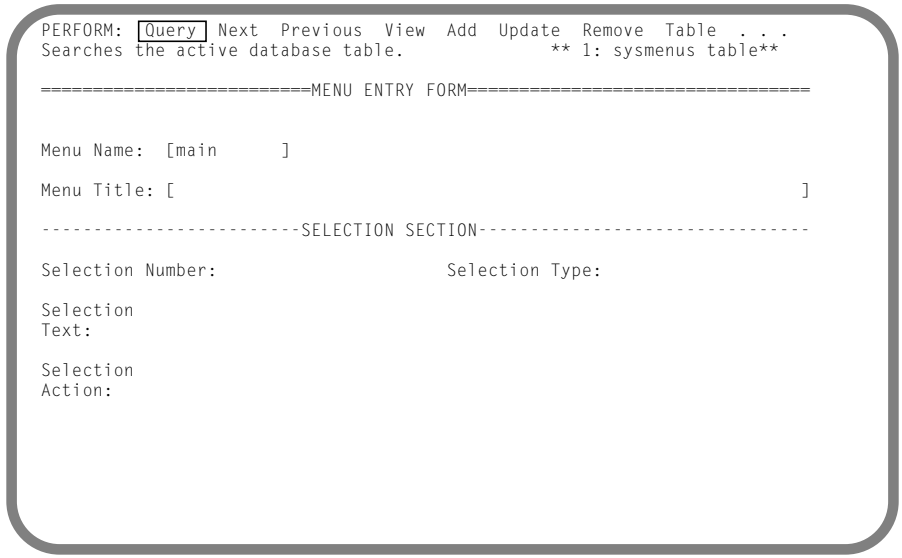
MENU NAME

INFORMIX-SQL uses the entry in the Menu Name field to find the menu you want when you make a selection that calls another menu. The menu name is used only by INFORMIX-SQL and never displays on a screen.

Usage

- The menu name must follow the standard rules for identifiers. It can be from 1 to 18 characters long; the first character must be a letter; and you can use numbers, letters, and underscores (_) for the rest of the name.
- The top-level menu must be named `main` in all lowercase letters.

The Menu Name entry for the Main menu must be `main`, as shown in the following screen.



MENU TITLE

Use this field to enter the text INFORMIX-SQL displays at the top of the menu.

Usage

The brackets on the screen show the maximum length of the text. It can contain any number of words that fit within the brackets.

The Menu Title entry for the Main menu follows.

```
PERFORM: [Query] Next  Previous  View  Add  Update  Remove  Table  . . .
Searches the active database table.              ** 1: sysmenus table**

==========================MENU ENTRY FORM=================================


Menu Name:  [main      ]

Menu Title: [WEST COAST WHOLESALERS, INC.                               ]

------------------------SELECTION SECTION-------------------------------

Selection Number:                     Selection Type:

Selection
Text:

Selection
Action:
```

SELECTION NUMBER

Use the Selection Number field to enter the option number you want to appear to the left of each menu item on the screen.

Usage

- INFORMIX-SQL displays the menu items in numbered order. The user selects items by number.
- The total number of options you can have in one menu depends on two factors: the number of lines your screen can hold and the length of the menu titles you enter. Most screens can accommodate 14 single-spaced menu lines, and each menu line can display 2 options of up to 33 characters. If the text for each option on a menu does not exceed 33 characters, you can display up to 14 double-spaced menu options (2 options per line) or 28 single-spaced menu options (2 options per line). If the text for an option is longer than 33 characters, it requires an entire line, reducing the total number of options available for that menu.

The Selection Number entry for an option on the REPORTS menu follows.

```
PERFORM: [Query] Next  Previous  View  Add  Update  Remove  Table  . . .
Searches the active database table.              ** 2: sysmenuitems table**

==========================MENU ENTRY FORM=================================

Menu Name:  [reports           ]

Menu Title:  REPORTS

-------------------------SELECTION SECTION--------------------------------

Selection Number:  [3      ]            Selection Type:  [S]

Selection
Text:     [RUN CUSTOMER MAILING LABELS                                  ]

Selection
Action:   [mailinglabels                                                ]
```

SELECTION TYPE

Use the Selection Type field to specify the type of action an option performs. You can indicate that an option runs a form or report; calls a menu; executes an INFORMIX-SQL command file, a program, or an operating system command; or invokes a script menu.

The following options are available for the Selection Type field.

Option	Purpose
F	Runs a form
R	Runs a report
M	Calls a menu
Q	Executes an INFORMIX-SQL command file
P	Executes a program or an operating system command
S	Executes a script menu

Usage

- The entry in the Selection Type field must agree with the entry in the Selection Action field. For example, when the Selection Type is R, the Selection Action must be the name of a compiled report.
- You can enter the Selection Type option in either an uppercase or lowercase letter. INFORMIX-SQL automatically displays it as an uppercase letter on the screen.

The Selection Type entry for running the **clist2** report follows.

```
PERFORM: [Query] Next  Previous  View  Add  Update  Remove  Table  . . .
Searches the active database table           ** 2: sysmenuitems table**

==========================MENU ENTRY FORM==================================

Menu Name:  [reports           ]

Menu Title:  REPORTS

------------------------SELECTION SECTION-------------------------------

Selection Number:  [2      ]            Selection Type:  [R]

Selection
Text:     [RUN REPORT ON CUSTOMER BY DESIGNATED STATE                  ]

Selection
Action:   [clist2                                                      ]
```

The Selection Type entry for running the **customer** entry form follows.

```
PERFORM: [Query] Next  Previous  View  Add  Update  Remove  Table  . . .
Searches the active database table           ** 2: sysmenuitems table**

==========================MENU ENTRY FORM==================================

Menu Name:  [forms             ]

Menu Title:  FORMS

------------------------SELECTION SECTION-------------------------------

Selection Number:  [1      ]            Selection Type:  [F]

Selection
Text:     [CUSTOMER ENTRY/QUERY FORM                                   ]

Selection
Action:   [customer                                                    ]
```

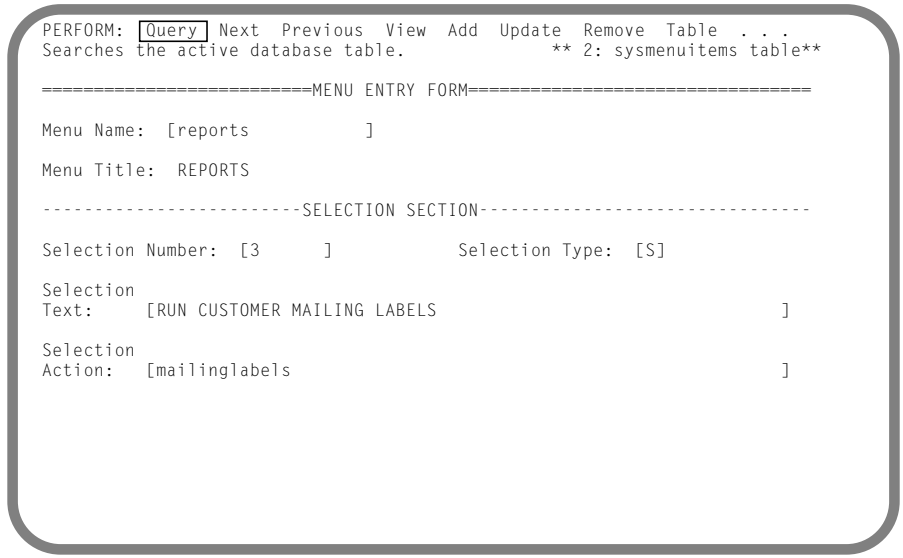
SELECTION TEXT

Use the Selection Text field to enter the text you want to appear to the right of the option number on the screen.

Usage

- The brackets on the screen show the maximum length of the text allowed in this field.
- The length of the selection text affects the total number of options you can include in a single menu. See the "Usage" section on page 5-19 for more information.

The Selection Text entry for an option on the REPORTS menu follows.



SELECTION ACTION

Use the Selection Action field to specify the name of the action executed when the user selects the option indicated in the Selection Type field. You can enter a compiled form or report specification, an INFORMIX-SQL command file, a menu, an operating system command, a program, or a script menu.

Usage

- The entry in the Selection Action field must agree with the entry in the Selection Type field, as the following table shows.

Selection Type	Selection Action
M	Enter the menu name (not the menu title) of the menu. You cannot enter a menu name that does not exist in the **Menu Name** field in the **sysmenus** table. Example: `reports`
P	Enter a program name or an operating system command. Example: date
F	Enter a form name. It is not necessary to add the .frm extension. Example: customer
PR	Enter a report name. It is not necessary to add the .arc extension. Example: clist1
Q	Enter an SQL command filename. It is not necessary to add the .sql extension. Example: `cust_city`
S	Enter a script menuname. Example: mailing label

- You can enter `Q` in the Selection Type field and nothing in the Selection Action field. When the user selects that option, INFORMIX-SQL calls the Query-language option on the INFORMIX-SQL Main menu. The user can then enter one or more SQL statements.

- You can enter an R or F in the Selection Type field and enter nothing in the Selection Action field. When the user selects this option, INFORMIX-SQL calls the Report or Form options, respectively, on the INFORMIX-SQL Main menu.
- When you finish using the Query-language Report or Form option, choose the E option to exit. INFORMIX-SQL then returns you to the USER-MENU menu.

The Selection Action entry used in the demonstration database for running the **clist1** report follows.

```
PERFORM: [Query] Next  Previous  View  Add  Update  Remove  Table  . . .
Searches the active database table.              ** 2: sysmenuitems table**

=========================MENU ENTRY FORM=================================

Menu Name:  [reports           ]

Menu Title:  REPORTS

------------------------SELECTION SECTION-------------------------------

Selection Number:  [1      ]            Selection Type:  [R]

Selection
Text:     [RUN CUSTOMER REPORT                                         ]

Selection
Action:   [clist1                                                      ]
```

The Selection Type R specifies that a report should be run. The Selection Action specifies **clist1** as the name of the report to be run.

Creating a Script Menu

A script menu includes more than one action for a single menu item. By selecting the appropriate menu option, these actions are run in sequence without the necessity of displaying the menu on the screen. An `S` in the Selection Type field requires the entry of a script menu name in the Selection Action field.

The following list describes the procedure used to create the **mailinglabels** script included in the demonstration user-menu. This script runs and displays customer mailing labels. It is Selection Number 3 on the REPORTS menu. When the user selects option 3 on the REPORTS menu, the mailing labels report runs, and the output file displays on the screen.

1. Select the Modify option on the USER-MENU menu. (See the section "Creating a Menu" on page 5-8.)
2. Type `a` to select the Add option.
3. Enter `reports` in the Menu Name field. The script becomes an option on the REPORTS Menu.
4. Enter `REPORTS` in the Menu Title field.
5. Press ESCAPE when you finish.

The preceding steps show you how to enter the necessary information in the **sysmenus** table. The following display shows how the PERFORM screen appears at this point.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  . . .
Added a row to the active database table.        ** 1: sysmenus table**

=========================MENU ENTRY FORM=================================

Menu Name:  [reports           ]

Menu Title: [REPORTS                                                   ]

------------------------SELECTION SECTION-------------------------------

Selection Number:                       Selection Type:

Selection
Text:


Selection
Action:




Row Added
```

The following steps show how to enter data into the lower half of the screen (the **sysmenuitems** table fields):

1. Type `d` to make the detail table active.
2. Type `a` to Add.
3. Enter `3` in the Selection Number field. The script becomes the third choice on the REPORTS menu.
4. Enter `S` in the Selection Type field. This indicates you will run a script menu.
5. Enter the following text in the Selection Text field:

   ```
   RUN CUSTOMER MAILING LABELS
   ```

   This text appears to the right of the option number on the screen.
6. Enter `mailinglabels` in the Selection Action field. This is the name of the script menu you want to run.
7. Press ESCAPE when you finish.

The following screen is the PERFORM screen as it appears at this point.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  . . .
Added a row to the active database table.        ** 2: sysmenuitems table**

==========================MENU ENTRY FORM==================================

Menu Name:  [reports           ]

Menu Title:  REPORTS

------------------------SELECTION SECTION-------------------------------

Selection Number:  [3      ]            Selection Type:  [S]

Selection
Text:     [RUN CUSTOMER MAILING LABELS
]

Selection
Action:   [mailinglabels                                            ]



Row added
```

You must now enter the actions you want the script to perform and the order in which you want them to be performed, as follows:

1. Type `m` for Master to make the **sysmenus** table active.
2. Type `a` to select the Add option.
3. Enter `mailinglabels` in the Menu Name field. This is the name of the script menu.
4. Enter the following text in the Menu Title field:

   ```
   run report menu selection 3 and display the output file
   ```

   Unlike all other user-menu menus, the entry in the Menu Title field for a script menu does not display on the screen. You can use it as a Comment line to list the series of actions that comprise the script menu.
5. Press ESCAPE when you finish.

The preceding steps show how to enter the necessary information in the **sysmenus** table. The following screen shows how the PERFORM screen appears at this point.

```
PERFORM:  Query  Next  Previous  View [Add] Update  Remove  Table  . . .
Adds a row to the active database table.          ** 1: sysmenus table**

==========================MENU ENTRY FORM=================================

Menu Name:  [mailinglabels     ]

Menu Title: [run report menu selection 3 and display the output file   ]

------------------------SELECTION SECTION-------------------------------

Selection Number:                       Selection Type:

Selection
Text:

Selection
Action:




Row added
```

The following steps show how to enter data for each option on the menu into the lower half of the screen (the **sysmenuitems** table fields):

1. Type `d` to make the detail table active.
2. Select the Add option.
3. Enter `1` in the Selection Number field. INFORMIX-SQL executes this action first.
4. Enter `R` in the Selection Type field. This runs a report.
5. Enter the following text in the Selection Text field:

   ```
   run mailing labels report
   ```

   Unlike other user-menu menus, the entry in the Selection Text field for a script menu does not display on the screen. You can use it as a Comment Line to describe the action specified by the entry in the Selection Type field.

6. Enter `mail` in the Selection Action field. This is the name of the compiled report specified by the action.
7. Press ESCAPE when you finish.

The following screen is the PERFORM screen as it appears at this point.

```
PERFORM:  Query  Next  Previous  View Add  Update  Remove  Table  . . .
Adds a row to the active database table.         ** 2: sysmenuitems table**

=========================MENU ENTRY FORM=================================

Menu Name:  [mailinglabels     ]

Menu Title:  run report menu selection 3 and display the output file

-------------------------SELECTION SECTION-------------------------------

Selection Number:  [1      ]            Selection Type:  [R]

Selection
Text:     [run mailing labels report                                    ]

Selection
Action:   [mail                                                         ]



Row added
```

You enter the second action for the **mailinglabels** script in a similar fashion to what you did when you set up the initial action, as follows:

1. Type `a` for Add.
2. Enter `2` in the Selection Number field. INFORMIX-SQL executes this action second.
3. Enter `P` in the Selection Type field. This executes a program.
4. Enter the following text in the Selection Text field:

   ```
   display output file from mailing labels report
   ```

5. Enter `type mail.out` in the Selection Action field. This is the name of the operating-system program you want to run.
6. Press ESCAPE when you finish.

The following screen is the completed PERFORM screen as it appears at this point.

```
PERFORM:  Query  Next  Previous  View [Add] Update  Remove  Table  . . .
Adds a row to the active database table.        ** 2: sysmenuitems table**

==========================MENU ENTRY FORM================================

Menu Name:  [mailinglabels     ]

Menu Title:  run report menu selection 3 and display the output file

------------------------SELECTION SECTION-------------------------------

Selection Number:  [2      ]            Selection Type:  [P]

Selection
Text:     [display output file from mailing labels report              ]

Selection
Action:   [type mail.out                                                ]



Row added
```

The two actions of running the report and displaying the output file are now entered as details of the **mailinglabels** script menu. When the user selects option 3 on the REPORTS menu, the **mailinglabels** script menu is selected, the mailing labels report is run, and the output file displays on the screen.

Use the Selection Type S when you want to run more than one action for a single menu item.

Chapter

6

Functions in ACE and PERFORM

In This Chapter

This chapter discusses calling C functions from ACE and PERFORM. While both ACE and PERFORM usually can handle all your database report and screen needs without modification, occasionally you might find it necessary to add a feature that is not present. For example, a C function called from ACE might make statistical computations on the data presented in a report and add these to the report. PERFORM might call C functions to check on the validity of data, to record the date, time, and name of the person updating the records, or to update the database.

The C functions can contain the following:

- Math functions or other C subroutines described in your system's C development manuals.
- If used with PERFORM, the functions described beginning on page 6-20.
- If used with INFORMIX-ESQL/C, the library routines and the INFORMIX-ESQL/C statements described in the *INFORMIX-ESQL/C Programmer's Manual*.

This chapter discusses the following topics:

- Calling C functions from ACE
- Calling C functions from PERFORM
- Writing the C program
- Using the PERFORM library functions
- Compiling, linking, and running customized versions of ACE and PERFORM

The chapter concludes with several examples.

Calling C Functions from ACE

The general format of an ACE report specification file includes the following seven sections:

- DATABASE section (required)
- DEFINE section
- INPUT section
- OUTPUT section
- SELECT section (SELECT or READ required)
- READ section (SELECT or READ required)
- FORMAT section (required)

You call a C function from within a report specification file by declaring the function name in the DEFINE section and by using the function in the FORMAT section. Then use ACEPREP to compile the report specification file.

FUNCTION

You declare a C function in the DEFINE section of the report specification file using the FUNCTION statement.

FUNCTION	is a required keyword.
userfunc	is the name by which the C function is referenced in the specification file. *userfunc* must satisfy the conditions for an ACE identifier.

Usage

You can declare several functions at the same time by repeating the keyword FUNCTION followed by the next function name. Do not include parentheses after the function name.

You can have PARAM, VARIABLE, and ASCII statements within the DEFINE section in addition to the FUNCTION statement.

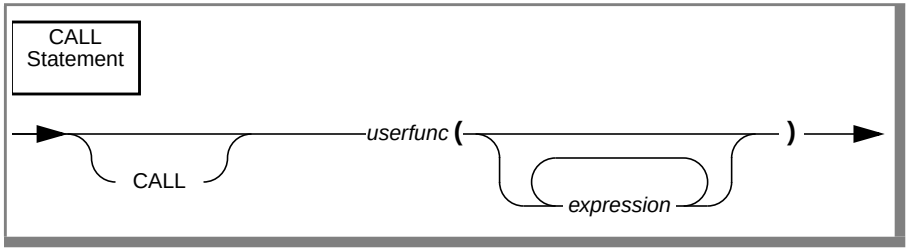
Calling C Functions

The FORMAT section of the report specification file contains one or more of the following control blocks that determine when ACE takes an action:

- PAGE HEADER control block
- PAGE TRAILER control block
- FIRST PAGE HEADER control block
- ON EVERY ROW control block
- ON LAST ROW control block
- BEFORE GROUP OF control block
- AFTER GROUP OF control block

Each control block contains one or more statements that tell ACE what action to take. For more information on the statements that ACE allows, see Chapter 4. In addition to the statements described in Chapter 4, you can use a C function call with the syntax shown on page 6-7.

CALL (in ACE)



CALL	is an optional keyword that you must use when the C function does not return a value. If you omit the CALL keyword, *userfunc* must return a value.
userfunc	is the name of a C function that you have previously declared in the DEFINE section.
expression	is 1 to 10 expressions, separated by commas.

Usage

An expression can include the following items:

- Numeric or character constants
- Column names
- ACE variables
- ACE parameters
- ACE functions (such as group aggregates and date functions)
- Quoted strings
- Arithmetic and logical operators
- Keywords

ACE statements are composed of keywords and expressions. You can use a C function in an expression wherever you can use a constant. When you use a function in this way you need not use the CALL keyword, but you must make sure the function returns a value.

IDS

If you are connecting to Informix Dynamic Server, you can pass columns of type VARCHAR, TEXT, or BYTE. You cannot, however, return TEXT or BYTE values from a C function. ♦

The following control block calls a C function **stat** that calculates statistics on the data in the rows that correspond to the **order_num** order:

```
after group of order_num
    call stat(order_num)
```

The following control block prints the order number and a value intended to correlate the total price of each order with the period of time the order has been outstanding. It calls a C function that computes the logarithm.

```
on every row
    print order_num,
    logarithm((total of total_price)/(today - order_date))
```

The following control block is taken from "ACE Example 1" on page 6-33. It prints the system date and time at the top of the first page of the report. The function **to_unix** sends its string argument to UNIX.

```
first page header
    call to_unix("date")
```

Compiling the Report Specification

Use ACEPREP to compile the report specification that includes calls to C function calls, just as you compile a report with no calls. When you name the file that contains the report specification file, assign it the .**ace** extension. For example, you could name the file **specfile.ace**. To invoke ACEPREP for this file, enter the following statement on the command line:

```
saceprep specfilefs
```

The **.ace** extension is optional when you identify the report specification file for the **saceprep** command.

For more information on compiling report specification files, see Chapter 4, "The ACE Report Writer."

Calling C Functions from PERFORM

The general format of a PERFORM form specification file includes the following five sections:

- DATABASE section (required)
- SCREEN section (required)
- TABLES section (required)
- ATTRIBUTES section (required)
- INSTRUCTIONS section (optional)

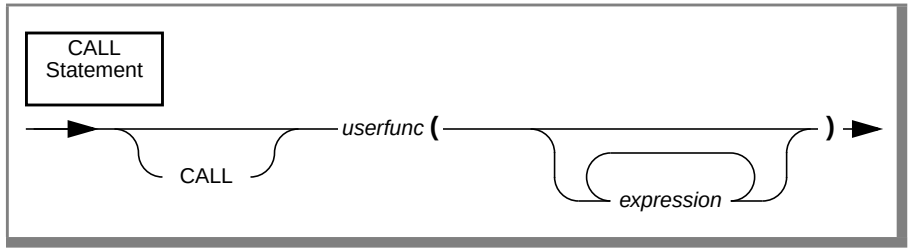
You can use C functions in the control blocks in the INSTRUCTIONS section of a form specification file. You can also use the ON BEGINNING and ON ENDING control blocks with a function call within the INSTRUCTIONS section.

Calling C Functions in the INSTRUCTIONS Section

In control blocks, you can use C functions anywhere you can use an expression, or the function can stand alone. Use the CALL statement for expressions or for simple function calls. The syntax of the CALL statement is described on page 6-10.

CALL (in PERFORM)

Use the CALL statement in PERFORM to execute a C function or to retrieve values from a C function.



CALL	is an optional keyword that you must use when the C function does not return a value. If you omit the keyword CALL, *userfunc* must return a value.
userfunc	is the name by which the C function is referenced in the PERFORM specification.
expression	is a list of 1 to 10 expressions. An expression is defined as follows: ■ A field tag ■ A constant value ■ An aggregate value ■ A C function ■ The keyword TODAY ■ The keyword CURRENT ■ Any combination of the preceding items, combined by using the arithmetic operators +, -, *, and /.

IDS

If you are connecting to Informix Dynamic Server, you can pass columns of type VARCHAR, TEXT, or BYTE. You cannot, however, return TEXT or BYTE values from a C function. ♦

The following examples demonstrate several methods of calling C functions:

```
after editadd of proj_num
    let f001 = userfunc(f002)
fs
before editupdate of paid_date
    if boolfunc(f003) then
        let f004 = 15
    else
        let f004 = 10

after add update remove of customer
    call userfunc()
```

ON BEGINNING and ON ENDING Control Blocks

You can use the ON BEGINNING and ON ENDING control blocks only with calls to C functions. Specify these control blocks in the INSTRUCTIONS section of the form specification file. ON BEGINNING executes immediately after invoking PERFORM, and ON ENDING executes immediately after the EXIT command.

For example, use ON BEGINNING to perform one of the following actions:

- Give instructions
- Request a special password
- Initialize a temporary work file in which to keep a batch of transaction records

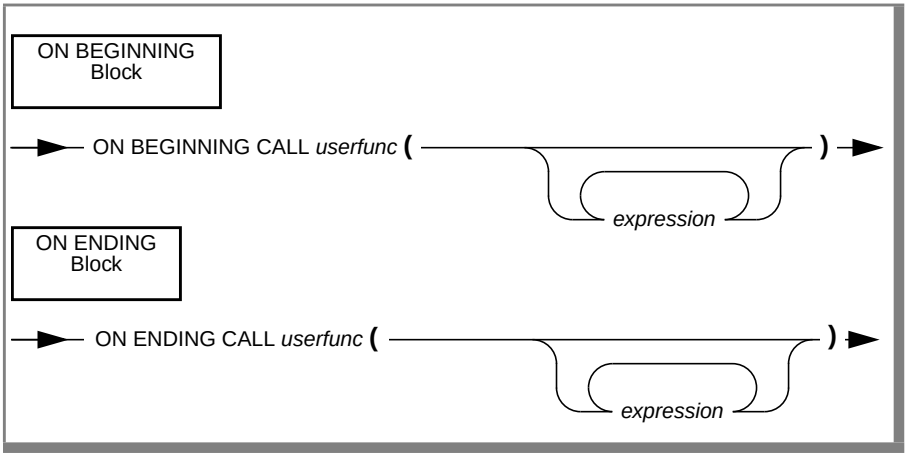
For example, use ON ENDING to perform one of the following actions:

- Perform calculations to summarize the changes made in the database during the PERFORM session that just concluded
- Print summaries of the records added
- Erase work files

You can include multiple ON BEGINNING and ON ENDING control blocks in the INSTRUCTIONS section. However, you can include only one CALL statement in each control block.

ON BEGINNING and ON ENDING

Use ON BEGINNING to designate that a C function call occurs immediately after you invoke PERFORM. Use ON ENDING to designate that a C function call occurs immediately after the EXIT command.



ON BEGINNING	are required keywords that begin the ON BEGINNING control block.
ON ENDING	are required keywords that begin the ON ENDING control block.
CALL	is a required keyword.
userfunc	is the name by which the C function is referenced in the PERFORM specification.
expression	is a list of 1 to 10 expressions. An expression is defined as one of the following items: ■ A field tag ■ A constant value ■ An aggregate value ■ A C function ■ The keyword TODAY ■ The keyword CURRENT ■ Any combination of the preceding items, combined by using the arithmetic operators +, -, *, and /

Compiling the Form Specification

Use FORMBUILD to compile the form specification file that includes calls to C functions just as you compile a report with no calls to C. When you name the form specification file, assign it the **.per** extension. For example, you could name the file **specfile.per**. To invoke FORMBUILD with this file, enter the following statement on the command line:

```
sformbld specfile
```

The **.per** extension is optional when you identify the form specification file for the **sformbld** command.

For more information on compiling form specification files, see Chapter 2, "The FORMBUILD Transaction Form Generator."

Writing the C Program

The C program must include the appropriate header files and structure declarations, as well as the C functions. This section describes the following topics:

- Organizing the C program
- Passing values to the C program
- Returning values to ACE and PERFORM

Organizing the C Program

To create a custom version of ACE or PERFORM that includes your functions, you must write a C program that contains the appropriate declarations. Your program can have one or more functions, and you can define other functions to use internally in your program. The following example illustrates the general structure of a C program that includes two user-defined functions:

```
#include "ctools.h"
/* add other includes as desired */

valueptr funct1();
valueptr funct2();

struct ufunc userfuncs[] =
```

```
{
"myfunct1", funct1,
"myfunct2", funct2,
0,0
};

/* add other global declarations */

valueptr funct1()
{
     .
     .
     .
/* funct1 takes no arguments
   and returns a character string    */
     .
     .
     .
strreturn(s, len);
}

valueptr funct2(arg1, arg2)
valueptr arg1, arg2;
{
.
.
.
/* funct2 takes two arguments
   and returns no value       */
.
.
.
}
```

The following steps describe how to organize your C program:

1. Place the **ctools.h** header file at the top of the program:

   ```
   #include "ctools.h"
   ```

   You might want to include other header files, such as **math.h** or **stdio.h**, depending on your application. If you use INFORMIX-ESQL/C, you can include **sqlca.h** and other header files.

   The **ctools.h** header file automatically includes the following additional header files:

   - **value.h**
   - **datetime.h**
   - **sqltypes.h**

2. Before you initialize the required array of **ufunc** structures, you must declare your functions. Included in **ctools.h** is the definition of the **value** structure and pointers to that structure, as shown in the following example:

```
typedef struct value *valueptr;
typedef struct value *acevalue;
typedef struct value *perfvalue;
```

   The last two pointers are included for compatibility with earlier releases of ACE and PERFORM. All your functions must be of type **valueptr**. If **funct1( )** and **funct2(arg1, arg2)** are your functions, declare them next:

```
valueptr funct1();
valueptr funct2();
```

3. Make the structure declaration and initialization for **userfuncs[ ]** the next section of your program. This structure is required so that ACE and PERFORM can call your functions at run time:

```
struct ufunc userfuncs[] =
    {
    "myfunct1", funct1,
    "myfunct2", funct2,
    0,0
    };
```

   The quoted strings, `"myfunct1"` and `"myfunct2"`, must be the names of the functions as they appear in the specification file. **funct1** and **funct2** (which correspond to `"myfunct1"` and `"myfunct2"`), are pointers to the functions as defined within the C program. Note that the C functions do not need to have the same names that you used in your specification file. The purpose of the **userfuncs** array is to make the connection between these two names. The two zeros at the end of the array are required as terminators.

4. The last section of the C program is the code for your functions. As stated earlier, all the functions that you call in ACE or PERFORM must be declared as returning pointers to a **value** structure. Also, all arguments of your functions must be declared type **valueptr**.

   Several macros are included that you can use to return values of type **valueptr**. These and other conversion routines are described in "Passing Values to a C Function" on page 6-16.

Passing Values to a C Function

Including the **ctools.h** header file allows you to pass values to the C functions from ACE and PERFORM. When passing values to a C function, the C function must determine the data type of the data passed. The C function has two options for determining the data type:

- Testing for the data type
- Converting the data type

Testing for the Data Type

By using the following definitions, you can test for the type of data passed to the C function. For example, if the parameter passed to the C function is **arg**, you can use the following definitions to detect the data type of **arg** and to extract the value of **arg**.

Definition	Returns
arg->v_charp	pointer to string
arg->v_len	length of string
arg->v_int	integer value
arg->v_long	long value
arg->v_float	float value
arg->v_double	double value
arg->v_decimal	decimal, money, datetime, or interval value
arg->v_type	data type
arg->v_ind	null indicator
arg->v_prec	datetime / interval qualifier

You can determine the data type of **arg** by checking `arg->v_type` against a series of integer constants defined in **sqltypes.h**.

v_type	SQL Type	C Type
SQLCHAR	CHAR	char string fixchar
SQLSMINT	SMALLINT	short
SQLINT	INTEGER	long
SQLFLOAT	FLOAT	double
SQLSMFLOAT	SMALLFLOAT	float
SQLDECIMAL	DECIMAL	dec_t
SQLSERIAL	SERIAL	long
SQLDATE	DATE	long
SQLMONEY	MONEY	dec_t
SQLDTIME	DATETIME	dtime_t
SQLINTERVAL	INTERVAL	intrvl_t

If `arg->v_type` is SQLCHAR, then the pointer to the string is available in `arg->v_charp`, and the number of characters in the string (length) is available in `arg->v_len`. The string is *not* null-terminated.

`arg->v_ind` is set to a negative value if the value of **arg** is NULL; otherwise `arg->v_ind` is set to zero.

IDS

If you are connecting to Informix Dynamic Server, you can use the VARCHAR data type as well. You can check `arg->v_type` against the following in **sqltypes.h**.

v_type	SQL Type	C Type
SQLVCHAR	VARCHAR	char

You cannot pass TEXT or BYTE values. ♦

Converting the Data Type

The **ctools.h** header file provides an alternative to testing the type of the parameter passed from ACE or PERFORM. Several functions, listed in the following table, can force conversion of a parameter passed as a pointer to a **value** structure, to a C data type of your choice.

Function	Returned Type
toint	int
tolong	long
tofloat	double
todouble	double
todate	long
todecimal	dec_t
todatetime	dtime_t
tointerval	intrvl_t

All these functions require a pointer to a type **value** structure and return a value of the type indicated. The **todecimal**, **todatetime**, and **tointerval** functions each require a second argument, as the following table shows.

Function	Second Argument
todecimal	pointer to the dec_t structure
todatetime	pointer to the dtime_t structure
tointerval	pointer to the intrvl_t structure

If the type conversion is not successful, the global integer **toerrno** is set to a negative value; if the conversion is successful, **toerrno** is set to zero.

Returning Values to ACE and PERFORM

If a function returns a value to ACE or PERFORM, you must insert the value in a type **value** structure and return a pointer to that structure. The **ctools.h** header file contains the following macros to perform that procedure for you.

Macro	Type Returned
intreturn(i)	returns integer i
lngreturn(l)	returns long l
floreturn(f)	returns float f
dubreturn(d)	returns double d
strreturn(s,c)	returns string s of length c (short)
decreturn(d)	returns decimal d (of type dec_t)
dtimereturn(d)	returns datetime d (of type dtime_t)
invreturn(i)	returns interval i (of type intrvl_t)

Use the appropriate macro even when you want to return an error condition. Do not use a simple **return**.

Because **strreturn(s,c)** returns a pointer to the string **s**, be sure to define **s** as a static or external variable.

IDS

If you are connecting to Informix Dynamic Server, you can use the following macro to return VARCHAR values.

Macro	Type Returned
vcharreturn(s,c)	returns string s of length c (short)

You cannot return TEXT or BYTE values. ♦

PERFORM Library Functions

The following five C functions are designed to control PERFORM screens from within C functions.

Function	Purpose
pf_gettype()	determines the type and length of a display field
pf_getval()	reads a value from a display field
pf_putval()	puts a value onto a display field
pf_nxfield()	moves the cursor to a specified field
pf_msg()	writes a message at the bottom of the screen

These functions are described in detail on the following pages. If these functions execute successfully, they return 0; if they are unsuccessful, they return a non-zero error code.

PF_GETTYPE()

The **pf_gettype()** function returns the SQL data type and the length of the display field for a specified field tag.

```
pf_gettype(tagname, type, len)
    char *tagname;
    short *type, *len;
```

tagname is a string containing the field tag that specifies a display field.

type is a pointer to a short integer that describes the data type of the display field tagname.

len is a pointer to a short integer that is the length of the display field tagname on the PERFORM screen.

Usage

The options for *type* follow.

type	SQL Type
SQLCHAR	CHARACTER
SQLSMINT	SMALLINT
SQLINT	INTEGER
SQLFLOAT	FLOAT
SQLSMFLOAT	SMALLFLOAT
SQLDECIMAL	DECIMAL
SQLSERIAL	SERIAL
SQLDATE	DATE
SQLMONEY	MONEY
SQLDTIME	DATETIME
SQLINTERVAL	INTERVAL

IDS

If you are connecting to Informix Dynamic Server, you can also specify the following data types.

type	SQL Type
SQLVCHAR	VARCHAR
SQLTEXT	TEXT
SQLBYTE	BYTE

All these types are defined in the **sqltypes.h** header file. ♦

Return Codes

0 The operation was successful; display field was found.

3759 There is no such field tag in the form.

PF_GETVAL()

If the display field is a character field, **pf_getval()** obtains the value found in a display field and the length of the value.

```
pf_getval(tagname, retvalue, valtype, vallen)
    char *tagname, *retvalue;
    short  valtype, vallen;
```

tagname is a string containing the field tag that specifies a display field.

retvalue is a pointer to the string, short, long, float, double, decimal, datetime, or interval structure returned by **pf_getval()**.

valtype is a short integer indicating the type of value to which retvalue should point.

vallen is a short integer specifying the length of the string (plus 1 for the terminating null byte) returned in retvalue, when *valtype* is CCHARTYPE. For any other value for *valtype*, *vallen* is ignored.

Usage

The parameter *retvalue* must be a pointer to the variable that contains the value. A common programming error is to use the variable itself. This results in a run-time system error and is not detected by the compiler.

The options for *valtype* are as follows.

valtype	SQL Type
CCHARTYPE	
CFIXCHARTYPE	CHARACTER
CSTRINGTYPE	
CINTTYPE	INTEGER
CSHORTTYPE	SMALLINT
CLONGTYPE	INTEGER, DATE, SERIAL

(1 of 2)

valtype	SQL Type
CFLOATTYPE	SMALLFLOAT
CDOUBLETYPE	FLOAT
CDECIMALTYPE	DECIMAL, MONEY
CDATETIME	DATETIME
CINTERVAL	INTERVAL

(2 of 2)

The value given to the parameter *valtype* determines the type of *retvalue*. The parameter *valtype* need not correspond exactly to the data type of the display field, but both should be either a number or a character so that PERFORM can do the proper type conversion.

If *valtype* is a number field and the display field is a character field, INFORMIX-SQL tries to convert the data type of *valtype*. If the conversion is unsuccessful, *retvalue* points to a zero. If *valtype* is character and the display field is a number field, a conversion to a string occurs. If the string does not fit in the length specified by *vallen*, *retvalue* contains the string, truncated to fit and null-terminated.

IDS

If you are connecting to Informix Dynamic Server, you can also specify the following data types for *valtype*.

valtype	SQL Type
CVCHARTYPE	SQLVCHAR
CLOCATORTYPE	SQLTEXT
	SQLBYTES

For VARCHAR values, *vallen* must contain the number of bytes the value buffer can hold. For TEXT and BYTE values, if you point *retvalue* to a **loc_t** structure, PERFORM copies the internal locator of **loc_t** to your structure. ♦

Return Codes

0	The operation was successful; display field was found.
3700	The user is not permitted to read the field.
3759	There is no such field tag in the form.

PF_PUTVAL ()

The **pf_putval()** function puts a value into a PERFORM screen in a specified display field. The user must have permission to update or to enter data into the desired destination field.

```
pf_putval(pvalue, valtype, tagname)
    char *pvalue;
    short  valtype;
    char *tagname;
```

pvalue is a pointer to a string, short, integer, long, float, double, decimal, datetime, or interval structure inserted into the display field designated by tagname.

valtype is a short integer indicating the type of the value to which pvalue points.

tagname is a string containing the field tag that specifies the display field where the information pointed to by pvalue is placed.

Usage

The *pvalue* parameter must be a pointer to the variable containing the value. A common programming error is to use the variable itself. This results in a run-time system error and is not detected by the compiler.

The options for *valtype* are as follows.

valtype	SQL Type
CCHARTYPE	
CFIXCHARTYPE	CHARACTER
CSTRINGTYPE	
CINTTYPE	INTEGER
CSHORTTYPE	SMALLINT
CLONGTYPE	INTEGER, DATE

(1 of 2)

valtype	SQL Type
CFLOATTYPE	SMALLFLOAT
CDOUBLETYPE	FLOAT
CDECIMALTYPE	DECIMAL, MONEY
CDATETIME	DATETIME
CINTERVAL	INTERVAL

(2 of 2)

If *valtype* is one of the character types and the display field is a number field, PERFORM tries to convert *valtype*. If the conversion is unsuccessful, PERFORM enters 0 in the display field.

If the type specified is a number field and the display field is character, a conversion to a string occurs. If the string does not fit in the display field, PERFORM truncates the display field.

If a number value does not fit in a number display field, PERFORM fills the field with asterisks.

IDS

If you are connecting to Informix Dynamic Server, you can also specify the following data types for *valtype*.

valtype	SQL Type
CVCHARTYPE	SQLVCHAR
CLOCATORTYPE	SQLTEXT
	SQLBYTES

Use this function with VARCHAR values just as you use it with CHARACTER values.

If you use this function with TEXT or BYTE data types, *pvalue* must point to a **loc_t** structure. PERFORM requires that the **loc_t** structure contain exactly the same information as the **loc_t** structure corresponding to *tagname*. For this reason, refrain from changing anything in your copy of the locator. You can then use the locator to change the actual value of the TEXT or BYTE data type, which PERFORM stores in a temporary file.♦

Return Codes

0	The operation was successful; display field was found.
3710	The user is not permitted to update the field.
3720	The user is not permitted to add to the field.
3756	The display field is not in the current table.
3759	There is no such field tag in the form.

PF_NXFIELD ()

The **pf_nxfield()** function controls the cursor placement on a PERFORM screen when you add a new record or update an old record.

```
pf_nxfield(tagname)
     char *tagname;
```

tagname is a string that contains the field tag for the display field on a PERFORM screen to which the cursor is sent.

Usage

The following list describes what happens at the different times when you call **pf_nxfield**:

- If called during a BEFORE EDITADD or a BEFORE EDITUPDATE of a table, it controls which display field is edited first.
- If called during an AFTER EDITADD or an AFTER EDITUPDATE of a table, it causes the cursor to move to the designated display field *tagname* for further editing, rather than allowing PERFORM to write the record.
- If called either BEFORE or AFTER an EDITADD or EDITUPDATE of a column, it determines the next field to be edited.
- If called either AFTER ADD or AFTER UPDATE, it is inoperative, since the record has already been written.

If *tagname* is set equal to the value EXITNOW, **pf_nxfield** causes an immediate exit from the add or update operation with the row being added or updated. This option performs the same as when you press ESCAPE to complete the transaction.

Return Codes

0	The operation was successful; display field was found.
3710	The user is not permitted to update the field.
3720	The user is not permitted to add to the field.
3755	The display field is display-only.
3756	The display field is not in the current table.
3759	There is no such field tag in the form.

PF_MSG()

The **pf_msg()** function displays a message at the bottom of the screen.

```
pf_msg(msgstr, reverseflag, bellflag)
    char *msgstr;
    short reverseflag, bellflag;
```

msgstr is a string containing the message displayed at the bottom of the screen.

reverseflag is a short integer indicating whether the message is displayed in reverse video. 0 indicates normal video; 1 indicates reverse video.

bellflag is a short integer indicating whether the terminal bell is rung when the message is displayed. 0 indicates not to ring the bell; 1 indicates to ring the bell.

Usage

If several calls to **pf_msg** are invoked at the same time in response to satisfying several conditions simultaneously, only the last message displayed is visible to the user.

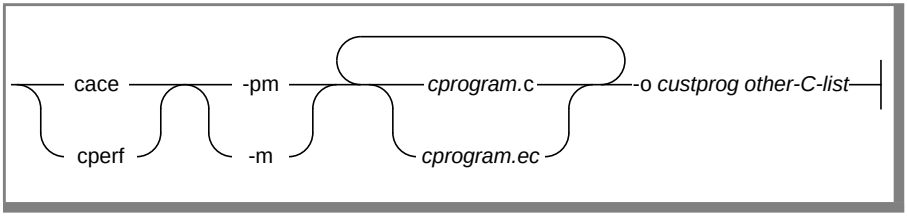
In normal video display, *msgstr* can have up to 80 characters. In reverse video display, the maximum number of characters is less than 80 because the reverse video control characters require one or more spaces on some monitors.

Compiling, Linking, and Running Reports and Forms

After you have written the file containing your C functions, you must compile the files and link the necessary library functions to create a custom version of **sacego** or **sperform**.

INFORMIX-SQL provides programs to simplify the compiling and linking process. You do not need to be concerned with names of special ACE or PERFORM libraries, nor with the location of the include files associated with these programs; the **cace** and **cperf** programs include these files automatically.

Syntax of the cace and cperf programs



cace	is the program that creates a custom version of **sacego**.
cperf	is the program that creates a custom version of **sperform**.
cprogram	is the name of the C program that contains your functions, as described in the previous sections.
.C	is the extension to use if *cprogram* only contains C statements.
.EC	is the extension to use if you have INFORMIX-ESQL/C and *cprogram* includes any INFORMIX-ESQL/C statements.
-m	compiles real-mode application—medium model only
-pm	create protected-mode application—large model only
-o	specifies the output filename.
custprog	is the name of your custom version of **sacego** or **sperform**.
other-C-list	is the rest of the arguments that you want to pass to the standard **cc** program.

Use of cace and cperf

You can compile several C programs at the same time.

After you compile your custom version of **sacego** or **sperform**, you can run reports or forms with the following command line:

```
custprog specfile
```

where *custprog* is the output file of the **cace** or **cperf** command, and *specfile* is the name of the report or form specification file you compiled using ACEPREP or FORMBUILD. When using **sacego**, specify the **.arc** suffix for *specfile*; when using **sperform**, specify the **.frm** suffix for *specfile*.

Examples

This section contains examples of both ACE applications and PERFORM applications. ACE C functions can be used with PERFORM as well. These sample programs are delivered with the demonstration database.

ACE Example 1

The following specification file calls a user function to execute a system command. The program is named **a_ex1.ace** in the demonstration database.

```
database
    stores
end

define
    function to_unix
end

select * from customer
end

format
    first page header
        call to_unix("date")
        skip 1 line
    on every row
        print customer_num, 3 spaces,
            fname clipped, 1 space, lname
end
```

The function **to_unix.c** follows:

```
#include "ctools.h"

valueptr to_unix();

struct ufunc userfuncs[] =
{
"to_unix", to_unix,
0,0
};

valueptr to_unix(string)
valueptr string;
{
char  savearea[80];

/*copy bytes from string to savearea*/
bycopy(string->v_charp, savearea, string->v_len);

/*put null on end*/
savearea[string->v_len]=0;

system(savearea);
}
```

To execute this example, perform the following steps:

1. Compile the report by executing the following command:

   ```
   saceprep a_ex1.ace
   ```

2. Create a custom version of **sacego** by executing the following command:

   ```
   cace to_unix.c -o output_file
   ```

   where *output_file* is the name of the file to contain the customer version of **sacego**.

3. Run the program by executing the following command:

   ```
   output_file a_ex1.ace
   ```

   where *output_file* is the name of the file containing the customer version of **sacego**.

ACE Example 2

The following ACE program computes the average and the standard deviation of the total cost of all the orders in the **stores7** demonstration database. This program is named **a_ex2.ace** in the demonstration database.

```
database
    stores
end

define
    function decsqroot
end

select o.order_num, sum(total_price) t_cost
    from orders o, items i
    where o.order_num = i.order_num
    group by o.order_num
end

format
    on every row
        print order_num, t_cost
    on last row
        skip 1 line
        print "The average total order is    : ",
            (total of t_cost)/count
            using "$#####.##"
        print "Standard deviation is         : ",
            decsqroot((total of t_cost*t_cost)/count
                - ((total of t_cost)/count)**2)
            using "$#####.##"
end
```

The function **decsqrt.c** follows:

```
#include "ctools.h"
#include <math.h>

valueptr squareroot();

struct ufunc userfuncs[] =
    {
    "decsqroot", squareroot,
    0, 0
    };

valueptr squareroot(pnum)
valueptr pnum;
{
double dub;
dec_t  dec;
```

```
    /* convert decimal to double */
        dectodbl(&pnum->v_decimal, &dub);

        dub = sqrt(dub);

    /* convert double to decimal */
        deccvdbl(dub, &dec);

    /* return decimal */
        decreturn(dec);
    }
```

To execute this example, perform the following steps:

1. Compile the report by executing the following command:

   ```
   saceprep a_ex2.ace
   ```

2. Create a custom version of **sacego** by executing the following command:

   ```
   cace decsqrt.c -o output_file -lm
   ```

   where *output_file* is the name of the file to contain the customer version of **sacego**. You must specify **-lm** to include the math libraries.

3. Run the program by executing the following command:

   ```
   output_file a_ex2.ace
   ```

   where *output_file* is the name of the file containing the customer version of **sacego**.

PERFORM Example

This example demonstrates accessing and displaying the following data from UNIX:

- The current user's login
- The time the user entered some data

The sample program then displays the data on a form.

The following form specification file uses the **customer** table to let you enter new customers into the **stores7** database. The form also includes two DISPLAYONLY fields that display the name of the entry clerk and the entry time. (To include the name of the entry clerk and the entry time in the database, you would need to add entry clerk and entry time columns to the **customer** table, rather than use the DISPLAYONLY fields.)

The cursor moves from the upper left down through the Customer Data by following the order of the fields listed in the ATTRIBUTES section. After the Telephone field, the cursor moves to the Owner Name field. When the entry clerk presses ESCAPE to complete the transaction, PERFORM calls the C function **stamptime()**.

The form is in **p_ex1.per** in the demonstration database; **stamp.c** contains the function **stamptime()**.

```
database stores7
screen
{

    ************************************************************
    *                    Customer Form                         *
    *==========================================================*
    * Number      :[f000       ]                               *
    * Owner Name  :[f001           ][f002           ]          *
    * Company     :[f003                ]                      *
    * Address     :[f004                ]                      *
    *              [f005                ]                      *
    * City        :[f006           ] State:[a0] Zipcode:[f007 ]  *
    * Telephone   :[f008             ]                         *
    ************************************************************
    * Entry Clerk :[f009      ]       Time Entered :[f010      ] *
    ************************************************************
}

tables
    customer

attributes
f000 = customer.customer_num, noentry;
f001 = customer.fname;
f002 = customer.lname;
f003 = customer.company;
f004 = customer.address1;
f005 = customer.address2;
f006 = customer.city;
a0 = customer.state, default="CA", upshift, autonext;
f007 = customer.zipcode, autonext;
f008 = customer.phone;
f009 = displayonly type char;
f010 = displayonly type char;

instructions

after editadd editupdate of phone
    nextfield = f001

after editadd editupdate of customer
    call stamptime()

end
```

The function **stamptime()**, called by the form specification file when the entry clerk presses ESCAPE to complete the transaction, follows. In addition to the special function **pf_putval()** defined earlier in this section, **stamptime()** uses the system functions **time()**, **localtime()**, and **getlogin()**. The login name of the order taker is obtained from the string function **getlogin()** and is displayed in the screen field Entry Clerk.

The system time is decomposed into hours and minutes and then reconstructed into a string variable displayed in the screen field Time Entered. PERFORM then writes the record to the **customer** table, using the data on the screen.

```
#include <stdio.h>
#include <time.h>
#include "ctools.h"

valueptr stamptime();

struct ufunc userfuncs[] =
    {
    "stamptime", stamptime,
    0,0
    };

valueptr stamptime()
{
long seconds, time();
char usertime[10], *getlogin();
struct tm *timerec, *localtime();

seconds = time((long *) 0);
timerec = localtime(&seconds);

pf_putval(getlogin(), CCHARTYPE, "f009");

sprintf(usertime, "%02d:%02d",
    timerec->tm_hour, timerec->tm_min);
pf_putval(usertime, CCHARTYPE, "f010");
}
```

To execute this example, perform the following steps:

1. Compile the form by executing the following command:

   ```
   sformbld p_ex1.per
   ```

2. Create a custom version of **sperform** by executing the following command:

   ```
   cperf stamp.c -o output_file
   ```

   where *output_file* is the name of the file to contain the customer version of **sperform**.

3. Run the program by executing the following command:

   ```
   output_file p_ex1.frm
   ```

   where *output_file* is the name of the file containing the customer version of **sperform**.

Appendix

A

The Demonstration Database and Examples

The **stores7** demonstration database contains a set of tables that describe an imaginary business. You can access the data in the **stores7** demonstration database using the sample programs that appear in this book, as well as through application programs that are listed in the documentation of other Informix products. The **stores7** demonstration database is not MODE ANSI.

For a description of the tables and data in the **stores7** database, see the *Informix Guide to SQL: Reference*.

This appendix contains the following sections:

- Instructions for creating the demonstration database and examples.
- Listings for the sample forms and reports used in this manual and the *INFORMIX-SQL User Guide*.

Creating the Demonstration Database

To use the demonstration database and examples, you need to copy them. To do this, first create a directory to store the database and examples and make it your current working directory. Then, from the operating-system command line, enter:

```
isqldemo
```

The **isqldemo** program places the **stores7** demonstration database files in your current directory. It also copies all the sample reports, forms, and command files into your current directory.

SE

If you list the contents of your current directory, you will see filenames similar to the following ones (if you are using an INFORMIX-SE server, you will also see a directory named **stores.dbs** that contains files for the sample database.) ♦

c_index.sql	sample.frm	mail2.ace	ex9.sql
c_custom.sql	clist1.ace	mail2.arc	ex10.sql
c_items.sql	clist2.arc	mail3.ace	ex11.sql
c_orders.sql	clist2.ace	mail3.arc	ex12.sql
c_manuf.sql	clist2.arc	ex1.sql	ex13.sql
c_state.sql	ord1.ace	ex2.sql	ex14.sql
c_stock.sql	ord1.arc	ex3.sql	ex15.sql
c_stores.sql	ord2.ace	ex4.sql	ex16.sql
customer.per	ord2.arc	ex5.sql	ex17.sql
customer.frm	ord3.ace	ex6.sql	ex18.sql
orderform.per	ord3.arc	ex7.sql	ex19.sql
orderform.frm	maill.ace	ex8.sql	
sample.per	mail1.arc		

Additional forms, reports, and command files are included that are not part of the examples described in the manuals. These provide further opportunities for practice after you become familiar with the demonstration database and examples.

Restoring the Demonstration Database

As you work with your copy of the demonstration database, you can make changes in such a way that the illustrations in this manual no longer reflect what you actually see on your screen. This can happen if you enter a great deal of new information into the **stores7** demonstration database, delete the information that came with the database, or significantly alter the structure of the tables, forms, reports, or command files.

You can restore the demonstration database to an original condition (the one upon which the examples are based) by re-creating the database with the `isqldemo` command.

For example, you might want to make a fresh copy of the demonstration database each time you start a new chapter. The files that make up the demonstration database are protected so that you cannot make any changes to the original copy.

Sample PERFORM Form Specifications

This section gives the complete listings for the sample form specifications included with your software.

The *customer* Specification

```
database
stores7

screen
{


_

CUSTOMERS

Customer Number:  [f000       ]

       Company :  [f001                ]
     First Name:  [f002           ]     Last Name:   [f003            ]

       Address :  [f004                ]
                  [f005                ]

          City :  [f006           ]     State :  [a0]    Zip :  [f007 ]

     Telephone :  [f008              ]



}
end

tables
         customer

attributes

f000 = customer_num;
f001 = company, reverse;
f002 = fname, comments = "Please enter first name if available";
f003 = lname;
f004 = address1;
f005 = address2;
f006 = city;
a0   = state, upshift, autonext, include = ("CA", "OR", "NV", "WA");
       comments = "Legal states are CA, OR, NV, or WA";
f007 = zipcode, autonext;
f008 = phone, picture = "###-###-####XXXXXX";

end
```

The *orderform* Specification

```
database stores7

screen
{
================================================================================
CUSTOMER INFORMATION:
Customer Number: [c1         ]             Telephone: [c10               ]
        Company: [c4                  ]
     First Name: [c2        ]             Last Name: [c3             ]
        Address: [c5                  ]
                 [c6                  ]
   City: [c7             ]  State: [c8]     Zip: [c9   ]

================================================================================
ORDER INFORMATION:
     Order Number: [o11         ]  Order Date: [o12       ]
     Stock Number: [i13        ]Manufacturer: [i16]
                  [manu_name      ]
        Quantity: [i18        ]
     Total Price: [i19               ]
SHIPPING INFORMATION:
     Customer P.O.: [o20       ]

          Ship Date: [o21       ]   Date Paid: [o22       ]
}

end

tables
customer orders
items manufact

attributes

c1 = *customer.customer_num = orders.customer_num;
c2 = fname,
       comments = "Please enter initial if available ";
c3 = lname;
c4 = company;
c5 = address1;
c6 = address2;
c7 = city;
c8 = state, upshift, autonext,
       include = ("CA","OR","NV","WA");
c9 = zipcode;
c10 = phone, picture = "###-###-####x#####";
o11 = *orders.order_num = items.order_num;
```

```
o12 = order_date,
       default = today;
i13 = items.stock_num;
i16 = items.manu_code , lookup manu_name = manufact.manu_name,
       joining *manufact.manu_code, upshift;

i18 = quantity, include = (1 to 100);
i19 = total_price;
o20 = po_num;
o21 = ship_date;
o22 = paid_date;

instructions

customer master of orders;
orders master of items;

end
```

The *sample* Specification

```
database stores7

screen
{
================================================================================
================================================================================
CUSTOMER INFORMATION:

Customer Number: [c1         ]

        Company: [c4                  ]
     First Name: [c2        ]              Last Name: [c3             ]

        Address: [c5                  ]
                 [c6                  ]

           City: [c7             ]  State: [c8]   Zip: [c9   ]
      Telephone: [c10              ]


================================================================================
================================================================================
}

screen

{
================================================================================
CUSTOMER NUMBER: [c1         ]           COMPANY: [c4                  ]

ORDER INFORMATION:
     Order Number: [o11         ]       Order Date: [o12          ]

     Stock Number: [i13        ]      Manufacturer: [i16]
      Description: [s14           ]                [m17            ]
             Unit: [s16           ]
                                                Quantity: [i18        ]
                                              Unitprice: [s15               ]
                                            Total Price: [i19               ]

SHIPPING INFORMATION:
 Customer P.O.: [o20       ]           Ship Charge: [d1        ]

       Backlog: [a]             Total Order Amount: [d2        ]
     Ship Date: [o21       ]
     Date Paid: [o22       ]
  Instructions: [o23                                      ]

}
end

tables
customer items stock
orders manufact

attributes
c1 = *customer.customer_num
   = orders.customer_num;
```

```
c2 = fname,
       comments = "Please enter initial if available";
c3 = lname;
c4 = company, reverse;
c5 = address1;
c6 = address2;
c7 = city;
c8 = state, upshift, autonext,
     include = ("CA","OR","NV","WA"),
     default = "CA" ;
c9 =  zipcode, autonext;
c10 = phone, picture = "###-###-####x####";
o11 = *orders.order_num = items.order_num;
o12 = order_date, default = today, format = "mm/dd/yyyy";
i13 = items.stock_num;
    = *stock.stock_num, noentry, noupdate, queryclear;
i16 = items.manu_code, lookup m17 = manufact.manu_name
       joining *manufact.manu_code, upshift, autonext;
    = *stock.manu_code, noentry, noupdate,
           upshift, autonext, queryclear;
s14 = stock.description, noentry, noupdate;
s16 = stock.unit_descr, noentry, noupdate;
s15 = stock.unit_price, noentry, noupdate;
i18 = items.quantity, include = (1 to 50),
    comments = "Acceptable values are 1 through 50" ;
i19 = items.total_price;
o20 = po_num, required,
    comments = "If no P.O. Number enter name of caller" ;
a   = backlog, autonext;
o21 = ship_date, default = today, format = "mm/dd/yyyy";
o22 = paid_date, format = "mm/dd/yyyy";
o23 = ship_instruct;
d1 =  displayonly type money;
d2 =  displayonly type money;

instructions

customer master of orders;
orders master of items;
composites <items.stock_num, items.manu_code>
      *<stock.stock_num, stock.manu_code>

before editadd editupdate of orders
   nextfield = o20

before editadd editupdate of items
   nextfield = i13
after editadd editupdate of quantity
   let i19 = i18 * s15
   nextfield = o11

after add update query of items

   if (total of i19) <= 100 then
      let d1 = 7.50
      else
      let d1 = (total of i19) * .04

   let d2 = (total of i19) + d1
```

```
after display of orders

   let d1 = 0
   let d2 = 0

end
```

Sample ACE Report Specifications

The *clist1* Specification

```
{ File: clist1.ace - Customer List Specification 1}

database
        stores7
end

output
        left margin 2
end

select
        customer_num,
        fname,
        lname,
        company,
        city,
        state,
        zipcode,
        phone
    from
        customer
    order by
        city
end

format

first page header
    print column 32, "CUSTOMER LIST"
    print column 32, "-------------"
    skip 2 lines
    print "NUMBER",
        column 9, "NAME",
        column 32, "LOCATION",
        column 54, "ZIP",
        column 62, "PHONE"
    skip 1 line

page header
    print "NUMBER",
        column 9, "NAME",
        column 32, "LOCATION",
        column 54, "ZIP",
        column 62, "PHONE"
    skip 1 line

on every row
    print customer_num using "####",
        column 9, fname clipped, 1 space, lname clipped,
        column 32, city clipped, ", " , state,
        column 54, zipcode,
        column 62, phone
```

```
on last row
    skip 1 line
    print  "TOTAL NUMBER OF CUSTOMERS:",
        column 30, count using "##"
end
```

The *clist2* Specification

```
{ File:  clist2.ace -  Customer List Specification 2 }

database
        stores7
end

define
    variable thisstate char(2)
end

input
    prompt for thisstate using
        "Enter state (use UPPER CASE) for which you wish a customer list:  "
end

output
    left margin 0
end

select
        customer_num,
        fname,
        lname,
        company,
        city,
        state,
        zipcode,
        phone
    from
        customer
    where
        state matches $thisstate
    order by
        zipcode,
        lname
end

format

    first page header
    print column 32, "CUSTOMER LIST"
    print column 32, "-------------"
    skip 2 lines
    print "Listings for the State of ", thisstate
    skip 2 lines
    print "NUMBER",
        column 9, "NAME",
        column 32, "LOCATION",
        column 54, "ZIP",
        column 62, "PHONE"
    skip 1 line

    page header
    print "NUMBER",
        column 9, "NAME",
        column 32, "LOCATION",
        column 54, "ZIP",
        column 62, "PHONE"
```

```
  skip 1 line

  on every row
  print customer_num using "####",
      column 9, fname clipped, 1 space, lname clipped,
      column 32, city clipped, ", " , state,
      column 54, zipcode,
      column 62, phone

on last row
  skip 2 lines
  print "Number of customers in ",thisstate, " is ", count using "<<<<&"
end
```

The *mail1* Specification

```
{file   mail1.ace

Mailing Label Specification - 1 }

{Customized report to print mailing labels.
The report will be sorted by zip code and
last name.  The use of PRINT commands in
the FORMAT section allow specific columns
to be printed on a line.
Blank lines will appear where data is
absent, and spaces appear next to
city field as not clipped as in
mail2.ace specification
}

database
stores7
end

select *
from customer
order by zipcode, lname
end

format
  on every row
     print fname, lname
     print company
     print address1
     print address2
     print city, ", " , state, 2 spaces, zipcode
     skip 2 lines
end
```

The *mail2* Specification

```
{mail2.ace   file

Mailing Label Specification - 2 }

{This improved report has an OUTPUT section added,
and uses nested IF statements.  }

database
stores7
end

output
  top margin 0
  bottom margin 0
  left margin 0
  page length 9
  report to "labels"
end

select
    fname, lname,
    company,
    address1,
    address2,
    city, state, zipcode
from customer
order by zipcode, lname
end

format
  on every row
    if (city is not null) and
      (state is not null) then
    begin
       print fname clipped, 1 space, lname
       print company
       print address1
       if (address2 is not null) then
          print address2
       print city clipped, ", " , state,
          2 spaces, zipcode
       skip to top of page
    end
end
```

The *mail3* Specification

```
{file   mail3.ace

Mailing Label Specification - 3 }

{This report prints 1-3 mailing labels across a page.
It stores the labels in character strings (array1,
array2, and array3) as it reads each row, and prints
the labels when it has read the proper number of rows.
At run time, you specify the number of labels (1-3)
that you want ACE to print across the page.  }

database
stores7
end


define
   variable name     char(75)              {holds first and last names}
   variable cstzp    char(75)              {holds city, state, and zip}
   variable array1   char(80)              {Array for name line}
   variable array2   char(80)              {Array for street line}
   variable array3   char(80)              {Array for city, state, and zipcode}
   variable start    smallint              {start of current label in array}
   variable finish   smallint              {end of current label in array}
   variable l_size   smallint              {label width}
   variable white    smallint              {spaces between each label}
   variable count1   smallint              {number of labels across page}
   variable i        smallint              {label counter}
end


input
   prompt for count1 using "Number of labels across page? [1-3] "
end

output
   top margin 0
   bottom margin 0
   left margin 0
   report to "labels.out"
end


select
      *
   from
      customer
   order by
      zipcode
end

format
   first page header                    {Nothing is displayed in this
                                         control block.  It just
```

```
                                        initializes variables that are
                                        used in the ON EVERY ROW
                                        control block.}
      let i = 1                        {Initialize label counter.}
      let l_size = 72/count1           {Determine label width (allow
                                        eight spaces total between labels).}
      let white = 8/count1             {Divide the eight spaces between
                                        the number of labels across the page.}

   on every row
      let name = fname clipped, 1 space, lname
      let cstzp = city clipped,
                  ", ",
                  state,
                  2 spaces, zipcode
      let finish = (i * l_size) + white
                                        {This section assigns names, }
      let start = finish - l_size
                                        {addresses, and zip codes to }
      let array1[start, finish] = name
                                        {arrays 1, 2, 3 until        }
      let array2[start, finish] = address1
                                        {i = the number of labels    }
      let array3[start, finish] = cstzp
                                        {across a page.              }

      if i = count1 then
         begin
            print array1 clipped        {Print the stored addresses.}
            print array2 clipped        {Use clipped to remove trailing}
            print array3 clipped        {spaces for quicker printing.}

            skip 1 line

            let array1 = " "            {Reset the arrays to spaces.}
            let array2 = " "
            let array3 = " "

            let i  = 1
         end
      else
         let i = i + 1

   on last row
      if i > 1 then                      {Print the last set of addresses}
         begin                           {if there were any left.}
            print array1 clipped
            print array2 clipped
            print array3 clipped
         end
end
```

The *ord1* Specification

```
{file   ord1.ace

Order Specification - 1 }

database
stores7
end

output
  report to "ordlist1"
end

select

  orders.order_num ordnum,
  order_date, customer_num,
  po_num, ship_date, ship_charge,
  paid_date,

  items.order_num, stock_num, manu_code,
  quantity, total_price

  from orders, items

  where orders.order_num = items.order_num

  order by ordnum
end

format

before group of ordnum
  print "Order number:  ", ordnum using "#####",
    " for customer number:  ", customer_num
    using "#####"
  print "Customer P.O. :  ", po_num,
    "  Date ordered:  ", order_date
  skip 1 line
  print "Stockno", column 20,
    "Mfcode", column 28, "Qty", column 38, "Price"

on every row
  print stock_num using "###", column 20,
    manu_code, column 28, quantity using "###",
    column 38, total_price using "$$$,$$$.&&"

after group of ordnum
  skip 1 line
  print 5 spaces, "Total amount for the order:  ",
    group total of total_price using "$$,$$$,$$$.&&"
  skip 3 lines

end
```

The *ord2* Specification

```
{file   ord2.ace

Order Specification - 2 }

database
stores7
end

output
   left margin 0
   report to "ordlist2"
end

select

   customer.customer_num custnum, fname,
     lname, company,

   orders.order_num ordnum, order_date,
     orders.customer_num, po_num, ship_date,
     ship_charge, paid_date,

   items.order_num, items.stock_num snum,
     items.manu_code, quantity, total_price,

   stock.stock_num, stock.manu_code,
     description, unit_price

 from  customer, orders, items, stock

 where customer.customer_num = orders.customer_num
   and orders.order_num = items.order_num
   and items.stock_num = stock.stock_num
   and items.manu_code = stock.manu_code

 order by custnum,  ordnum, snum

end

format

before group of custnum
  print "Orders for:  ", fname clipped, 1 space,
  lname print 13 spaces, company
  skip 1 line

before group of ordnum
  print "Order number:  ", ordnum using "#####"
  print "Customer P.O. :  ", po_num,
    "  Date ordered:  ", order_date
  skip 1 line
  print "Stockno", column 10, "Mfcode", column 18,
    "Description", column 38, "Qty", column 43,
    "Unit price" , column 55, "Total for item"

on every row
  print snum using "###", column 10, manu_code,
    column 18, description clipped, column 38,
```

```
      quantity using "###", column 43,
      unit_price using "$$$$.&&", column 55,
      total_price using "$$,$$$,$$$.&&"

  after group of ordnum

    skip 1 line
    print 4 spaces,
      "Shipping charges for the order:  ",
      ship_charge using "$$$$.&&"
    skip 1 line

    print 5 spaces, "Total amount for the order:  ",
      ship_charge + group total of
      total_price using "$$,$$$,$$$.&&"
    skip 3 lines

  after group of custnum
    skip 2 lines

  end
```

The *ord3* Specification

```
{file   ord3.ace

Order Specification - 3 }

database
stores7
end

define
  variable begin_date date
  variable end_date date
end

input
  prompt for begin_date using
   "Enter beginning date for report:  "
  prompt for end_date using
   "Enter ending date for report:  "
end

output
  left margin 0
  report to "ordlist3"
end

select

  customer.customer_num, fname, lname, company,

  orders.order_num ordnum, orders.customer_num,
  order_date, month(order_date) months,
  day(order_date) days, year(order_date) years,

  items.order_num, quantity, total_price

  from  customer, orders, items

  where customer.customer_num = orders.customer_num
    and orders.order_num = items.order_num and
    order_date between $begin_date and $end_date

  order by years, months, days, company, ordnum
end

format

first page header
  print column 10, "========================================================="
  print column 10, "                       DAILY ORDER REPORT"
  print column 10, "========================================================="
skip 1 line
  print column 15, "FROM:  ", begin_date
    using "mm/dd/yy",
    column 35, "TO:  ", end_date
    using "mm/dd/yy"
  print column 15, "Report run date:  ",
    today using "mmm dd, yyyy"
  skip 2 lines
```

```
  print column 2, "ORDER DATE", column 15,
    "COMPANY", column 35, "NAME",
    column 57, "NUMBER", column 65, "AMOUNT"

before group of days
  skip 2 lines

after group of ordnum
  print column 2, order_date, column 15,
    company clipped, column 35, fname clipped,
    1 space, lname clipped, column 55,
    ordnum using "####", column 60,
    group total of total_price using "$$,$$$,$$$.&&"

after group of days
  skip 1 line
  print column 21, "Total amount ordered for the day:  ",
    group total of total_price using "$$$$,$$$,$$$.&&"
  skip 1 line
  print column 15, "=================================================="

on last row
  skip 1 line
  print column 15, "=================================================="
  skip 2 lines
  print "Total Amount of orders:  ", total of
    total_price using "$$$$,$$$,$$$.&&"

page trailer
   print column 28, pageno using "page <<<<"

end
```

Appendix

B

Setting Environment Variables

Various *environment variables* affect the functionality of your Informix products. You can set environment variables that identify your terminal, specify the location of your software, and define other parameters.

Some environment variables are required and others are optional. For example, you must set—or accept the default setting for—certain UNIX environment variables.

For a discussion of specific environment variables, see the *Informix Guide to SQL: Reference*. For a detailed discussion of variables and settings for Global Language Support, see the *Informix Guide to GLS Functionality*. This chapter documents only the following specific variables not described in those manuals:

- **DBFORM** (specific for INFORMIX-SQL): Specifies a directory where menu form files reside.
- **DBFORMAT** (specific for Informix tools): Specifies the default format in which the user inputs, displays, or prints values of numeric and money data types.

SE

- **DBTEMP** (specific for INFORMIX-SE servers): Specifies the full pathname of the directory into which you want INFORMIX-SE to place its temporary files. ♦

Where to Set Environment Variables

You can set Informix and UNIX environment variables in the following ways:

- At the system prompt on the command line

  When you set an environment variable at the system prompt, you must reassign it the next time you log into the system.

- In a special shell file, as follows:

  .login or **.cshrc** for the C shell

  .profile for the Bourne shell or the Korn shell

  When you set an environment variable in your **.login**, **.cshrc**, or **.profile** file, it is assigned automatically every time you log into the system.

Important: *Check that you do not inadvertently set an environment variable differently in your* ***.login*** *and* ***.cshrc*** *C shell files.*

- In an environment-configuration file

  This is a common or private file where you can define all the environment variables that are used by Informix products. Using a configuration file reduces the number of environment variables that you must set at the command line or in a shell file.

  Use the **ENVIGNORE** environment variable to later override one or more entries in this file. Use the following Informix **chkenv** utility to check the contents of an environment configuration file, and return an error message if an environment variable entry in the file is bad or if the file is too large:

  ```
  chkenv filename
  ```

  The **chkenv** utility is described in the *Informix Guide to SQL: Reference*.

  The common (shared) environment-configuration file resides in **$INFORMIXDIR/etc/informix.rc**. The permission for this shared file must be set to 644. A private environment-configuration file must be stored in the user's home directory as **.informix** and must be readable by the user.

Tip: *The first time you set an environment variable in a shell or configuration file, before you work with your Informix product, you should log out and then log back in, "source" the file (*C *shell), or use "." to execute an environment-configuration file (Bourne or Korn shell). This allows the process to read your entry.*

How to Set Environment Variables

You can change default settings and add new ones by setting one or more of the environment variables recognized by your Informix product. If you are already using an Informix product, some or all the appropriate environment variables might already be set.

After one or more Informix products have been installed, enter the following command at the system prompt to view your current environment settings:

BSD UNIX: `env`

UNIX System V: `printenv`

Use standard UNIX commands to set environment variables. Depending on the type of shell you use, Figure B-1 shows how you set the fictional ABCD environment variable to *value*.

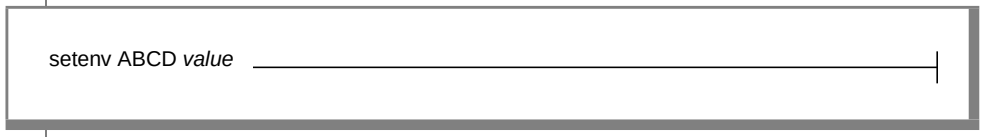
Figure B-1
Setting Environment Variables in Different Shells

C shell:	`setenv ABCD value`
Bourne shell *or* **Korn shell:**	`ABCD=value` `export ABCD`
Korn shell:	`export ABCD=value`

When Bourne-shell example settings are shown in this chapter, the Korn shell (a superset of the Bourne shell) is implied as well. Korn-shell syntax allows for a shortcut, as Figure B-1 shows.

Tip: *The environment variables are case sensitive.*

The following diagram shows how the syntax for setting an environment variable is represented throughout this chapter. These diagrams indicate the setting for the C shell; for the Bourne or Korn shell, follow the syntax in Figure B-1.



For more information on how to read syntax diagrams, see the introduction.

To unset most of the environment variables that this chapter shows, enter the following command:

C shell: `unsetenv ABCD`

Bourne shell *or* Korn shell: `unset ABCD`

Default Environment Variable Settings

The following list describes the main default assumptions that are made about your environment when you use Informix products. Environment variables used to change the specific default values are shown in parentheses. Other product-specific default values are described where appropriate throughout this chapter.

- The program, compiler, or preprocessor, and any associated files and libraries of your product have been installed in the **/usr/informix** directory.
- IDS SE The default Informix Dynamic Server or INFORMIX-SE database server for explicit or implicit connections is indicated by an entry in the **$INFORMIXDIR/etc/sqlhosts** file (**INFORMIXSERVER**). ♦
- The default directory for message files is **$INFORMIXDIR/msg** (**DBLANG** unset and **LANG** unset).
- SE If you are using INFORMIX-SE, the target or current database is in the current directory (**DBPATH**).

- Temporary files for INFORMIX-SE are stored in the **/tmp** directory (**DBTEMP**). ♦
- The default terminal-dependent keyboard and screen capabilities are defined in the **termcap** file in the **$INFORMIXDIR/etc** directory. (**INFORMIXTERM**)
- For products that use an editor, the default editor is the predominant editor for the operating system, usually **vi**. (**DBEDIT**)
- For products that have a print capability, the program that sends files to the printer is usually:

  lp for UNIX System V

  lpr for BSD and other UNIX systems (**DBPRINT**)

- The default format for money values is `$000.00`. (**DBMONEY** set to *$*.)
- The default format for dates is MM/DD/YYYY. (**DBDATE** set to MDY4*/*)
- The field separator for unloaded data files is the vertical bar (`|=ASCII 124`). (**DBDELIMITER** set to `|`)

Rules of Precedence

When an Informix product accesses an environment variable, normally the following rules of precedence apply:

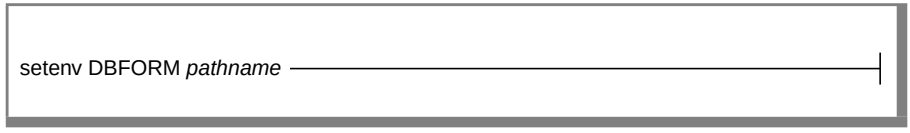
1. The highest precedence goes to the value as defined in the environment (shell).
2. The second-highest precedence goes to the value as defined in the private environment-configuration file in the user's home directory (**~/.informix**).
3. The next-highest precedence goes to the value as defined in the common environment-configuration file (**$INFORMIXDIR/etc/informix.rc**).
4. The lowest precedence goes to the default value.

Environment Variables

Important: *Most of the environment variables you need are described in the "Informix Guide to SQL: Reference" and the "Informix Guide to GLS Functionality." This appendix covers only environment variables not described in those manuals.*

DBFORM

The **DBFORM** environment variable specifies the subdirectory of **$INFORMIXDIR** (or full pathname) in which the menu form files for the currently active language reside. (**$INFORMIXDIR** means "the name of the directory referenced by the environment variable **INFORMIXDIR**"). Menu form files provide a set of language-translated menus to replace the standard INFORMIX-SQL menus. Menu form files have the suffix **.frm**. Menu form files are included in language supplements, which contain instructions specifying where the files should be installed and what **DBFORM** settings to specify.



pathname	specifies the subdirectory of $INFORMIXDIR or the full pathname of the directory that contains the message files.

Usage

If **DBFORM** is not set, the default directory for menu form files is **$INFORMIXDIR/forms**. The files should be installed in a subdirectory under the **forms** subdirectory under **$INFORMIXDIR**. For example, French menu files could be installed in **$INFORMIXDIR/forms/french** or in **$INFORMIXDIR/forms/fr.88591**. The English language version will normally be installed in **$INFORMIXDIR/forms** or **$INFORMIXDIR/forms/english**. Non-English menu form files should not be installed in either of the locations where English files are normally found.

Figure B-2 illustrates the search method employed for locating message files for a particular language (where the value set in the **DBFORM** environment variable is indicated as **$DBFORM**).

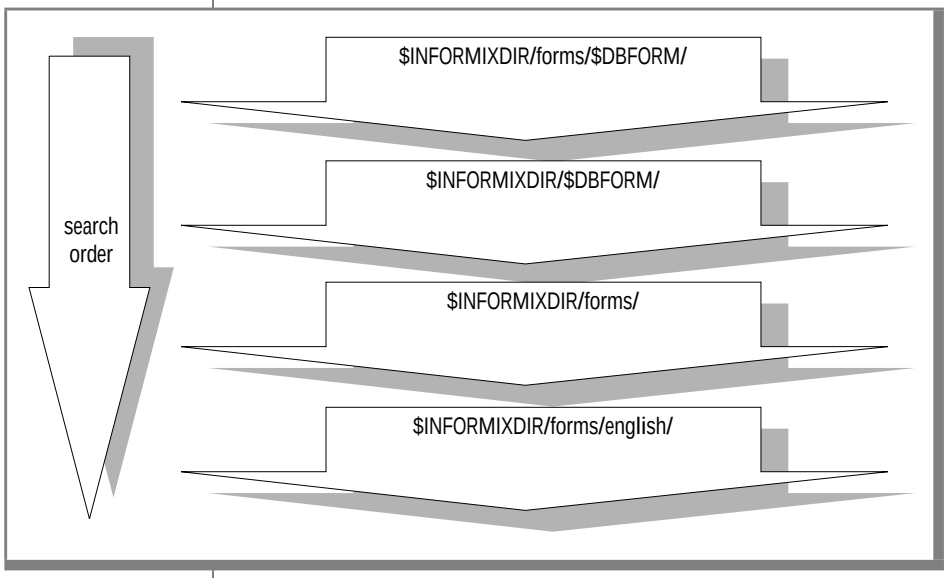


Figure B-2
Directory Search Order, Depending on $DBFORM

If the **LANG** variable is set, and **DBFORM** is not, the search order changes, as Figure B-3 shows.

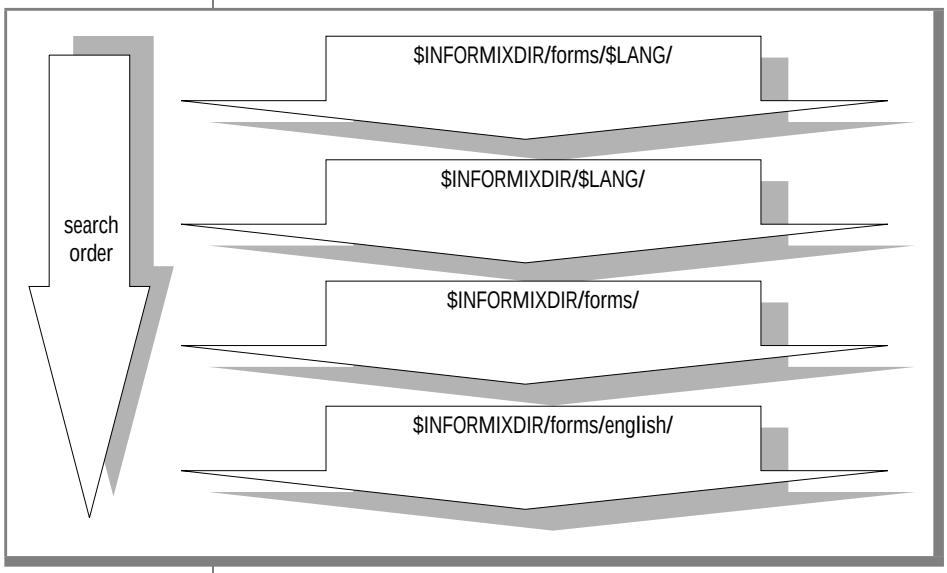


Figure B-3
Directory Search Order, Depending on $LANG

If both **DBFORM** and **LANG** are set, **LANG** is ignored in establishing search order.

To specify a menu form directory

1. Use the **mkdir** command to create the appropriate subdirectory in **$INFORMIXDIR/forms**.
2. Set the owner and group of the subdirectory to **informix** and the access permission for this directory to 755.
3. Set the **DBFORM** environment variable to the new subdirectory, specifying only the subdirectory name and not the full pathname.
4. Copy the **.frm** files to the new menu form directory specified by **$INFORMIXDIR/forms/$DBFORM**. All files in the menu form directory should have the owner and group **informix** and access permission 644.
5. Run your program or otherwise continue working with your product.

For example, you can store the set of menu form files for the French language in **$INFORMIXDIR/forms/french** as follows:

```
setenv DBFORM french
```

DBFORMAT

The Informix-defined **DBFORMAT** environment variable specifies the default format in which the user inputs, displays, or prints values of the following data types:

- DECIMAL
- FLOAT
- SMALLFLOAT
- INTEGER
- SMALLINT
- MONEY

The default format specified in **DBFORMAT** affects how numeric and monetary values are:

- Displayed and input on the screen
- Printed
- Input to and output from ASCII files using LOAD and UNLOAD

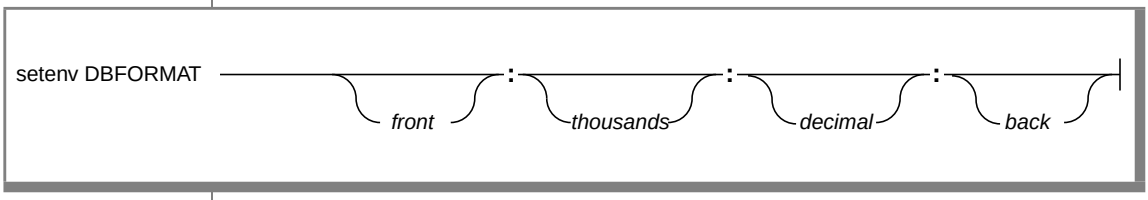
DBFORMAT is used to specify the leading and trailing currency symbols (but not their default positions within a monetary value) and the decimal and thousands separators. The decimal and thousands separators defined by **DBFORMAT** apply to both monetary and numeric data, and override the sets of separators established by GLS settings. For this reason, countries that use different formatting conventions for their monetary and numeric data should use GLS SETTINGS and avoid **DBFORMAT**. For more information on GLS, see Appendix C, "Global Language Support" and the *Informix Guide to GLS Functionality*.

The setting in **DBFORMAT** will affect the following INFORMIX-SQL keywords:

- USING expression in ACE
- FORMAT attribute in PERFORM

- PRINT statement in ACE
- LET statement in ACE, where a character string is receiving a monetary or numeric value

The syntax for setting **DBFORMAT** is as follows.



front	is the leading currency symbol. The front value is optional. The null string, represented by "*", is allowed, and means that the leading currency symbol is not applicable.
thousands	is a list of one or more characters that determine the possible thousands separator. The user can use any of the specified characters as the thousands separator when inputting values. The values in the list are not separated by spaces or other characters. INFORMIX-SQL uses the first value specified as the thousands separator when displaying the output value. You can specify any characters for the thousands separator except the following: ■ Digits ■ <, >, |, ?, !, =, [,] If you specify the * character, INFORMIX-SQL omits the thousands separator. The *thousands* value is optional. The default value is the *. A blank space can be the thousands separator and is used for this purpose in some locales. In versions prior to 6.0, the colon symbol (:) was not allowed as a thousands separator. In version 6.0, the colon symbol is permitted, but must be preceded by a backslash (\) symbol, as in the specification :\::.:DM.

(1 of 2)

decimal	is a list of one or more characters that determine the possible decimal separators. The user can use any of the specified characters as the decimal separator when inputting values. INFORMIX-SQL uses the first value specified as the decimal separator when displaying the output value. You can specify any characters except the following characters: ■ Digits ■ <, >, |, ?, !, =, [,] ■ Any characters specified for the *thousands* value The *decimal* value is optional. Specification of an asterisk symbol in the decimal position will cause displayed values not to have a decimal separator. The colon symbol is permitted as a decimal separator but must be preceded by a backslash (\) symbol in the **DBFORMAT** specification.
back	is a value that determines the trailing currency symbol. The back value is optional.

(2 of 2)

You must specify all three colons in the syntax. Enclosing the **DBFORMAT** specification in a pair of single quotes is suggested to prevent the shell from interpreting any of the characters.

Usage

The setting in **DBFORMAT** directly specifies the leading and trailing currency symbol, and the numeric and decimal separators. It adds the currency symbol and changes the separators displayed on the screen in a monetary or numeric field, and in the default format of a PRINT statement. For example, if **DBFORMAT** is set to:

```
*:.:,:DM
```

the value 1234.56 will print or display as:

```
1234,56DM
```

DM stands for deutsche marks. In the case of a screen form, values input by the user are expected to contain commas, not periods, as decimal separators if this **DBFORMAT** string has been specified.

The setting in **DBFORMAT** also affects the way format strings in the FORMAT attribute in ACE and the USING clause in PERFORM are interpreted. In these format strings, the period symbol (**.**) is not a literal character but a placeholder for the decimal separator specified by **DBFORMAT**. Likewise, the comma symbol (**,**) is a placeholder for the thousands separator specified by **DBFORMAT**. The dollar sign is a placeholder for the leading currency symbol. The at-sign (**@**) symbol is a placeholder for the trailing currency symbol. Figure B-4 illustrates the results of different combinations of **DBFORMAT** setting and format string on the same value.

Figure B-4
Illustration of the Results of Different DBFORMAT Settings and Format Strings

Value	Format String	DBFORMAT Setting	Displayed Result
1234.56	$$#,###.##	$:,::	$1,234.56
1234.56	$$#,###.##	:.:,:DM	1.234,56
1234.56	#,###.##@@	$:,::	1,234.56
1234.56	#,###.##@@	:.:,:DM	1.234,56DM

When the user enters values, INFORMIX-SQL behaves as follows:

- Disregards any currency symbols (leading or trailing) and thousands separators that the user enters.
- If a symbol appears that is defined as the decimal separator in **DBFORMAT**, it is interpreted in the input value as a decimal separator.

When INFORMIX-SQL displays or prints values:

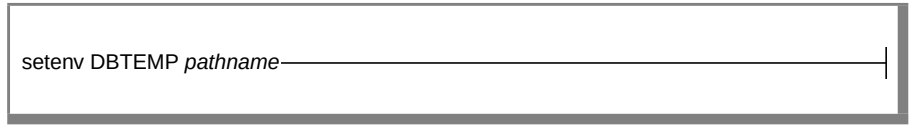
- The **DBFORMAT**-defined leading or trailing currency symbol is displayed for MONEY values.
- If a leading or trailing currency symbol is specified by the FORMAT attribute for non-MONEY data types, the symbol is displayed.
- The thousands separator does not display, unless it is included in a FORMAT attribute or USING operator.
- The decimal separator is displayed unless the decimal separator is defined as NULL (*) in **DBFORMAT** or the data type is integer (INT or SMALLINT).

When money values are converted to character strings using the LET statement in ACE, both the default conversion and the conversion with a USING clause will insert the **DBFORMAT**-defined separators and currency symbol into the created strings.

DBFORMAT, like **DBMONEY**, dictates both the numeric and monetary formats for data. In some countries, including Portugal and Italy, the correct use of decimal and thousands separators differs between numeric and monetary data. For such countries, GLS CONFIGURATION FILE SETTINGS provide for independently defined numeric and monetary formatting. This is in contrast to **DBFORMAT** and **DBMONEY**.

DBTEMP

Set the **DBTEMP** environment variable to specify the full pathname of the directory into which you want INFORMIX-SE to place its temporary files. You need not set **DBTEMP** if the default, **/tmp**, is satisfactory.



pathname	is the full pathname of the directory for temporary files.

Set the **DBTEMP** environment variable as follows to specify the pathname **usr/magda/mytemp**:

C shell:
```
setenv DBTEMP usr/magda/mytemp
```

Bourne shell:
```
DBTEMP=usr/magda/mytemp
export DBTEMP
```

For the creation of temporary tables, if **DBTEMP** is not set, the temporary tables are created in the directory of the database (that is, the **.dbs** directory).

Appendix

C

Global Language Support

This appendix identifies fundamental terms and concepts, as well as supported features, for Global Language Support (GLS) as implemented by INFORMIX-SQL.

Much of the information covered here is discussed in greater detail in the *Informix Guide to GLS Functionality*, primarily from the perspective of the database server. If you have no knowledge of GLS, consider reading that manual before using this appendix.

INFORMIX-SQL implements GLS in several areas:

- Entry, display, and editing of non-English characters
- References to SQL identifiers containing non-English characters
- Collation of strings containing non-English symbols
- Non-English formats for number, currency, and time values

Global Language Support Terms

GLS refers to the set of features that makes it possible to use non-Roman alphabets, diacritical marks, and so on. In order to understand the requirements of GLS, you will need to become familiar with the terms described in this section.

GLS is a set of features that enable you to use languages other than U.S. English. GLS includes the localized representation of dates, currency values, and numbers. INFORMIX-SQL supports the entry, retrieval, and display of multibyte characters in some East Asian languages, such as Japanese and Chinese.

Code Sets and Logical Characters

For a given language, the *code set* specifies a one-to-one correspondence between each logical element (called a *logical character*, or a *code point*) of the character set, and the bit patterns that uniquely encode that character. In U.S. English, for example, the ASCII characters constitute a code set.

Code sets are based on logical characters, independent of the font that a display device uses to represent a given character. The size or font in which INFORMIX-SQL displays a given character is determined by factors independent of the code set. (But if you select, for example, a font that includes no representation of the Chinese character for "star," then only whitespace will be displayed for that character, until you specify a font that supports it.)

Collation Order

Collation order is the sequence in which character strings are sorted. Database servers can support collation in either *code-set order* (the sequence of code points) or *localized order* (some other predefined sequence). For details of localized collation, see the *Informix Guide to GLS Functionality*.

INFORMIX-SQL supports only code-set order; the database server, rather than INFORMIX-SQL, must do the sorting if you require localized collation of data values in NCHAR or NVARCHAR columns of the database.

Single-Byte and Multibyte Characters

Most alphabet-based languages, such as English, Greek, and Tagalog, require no more than the 256 different code points that a single byte can represent. This simplifies aspects of processing character data in those languages; for example, the number of bytes of storage that an ASCII character string requires has a linear relationship to the number of characters in the string.

In non-alphabetic languages, however, the number of different characters can be much greater than 256. Languages like Chinese, Japanese, and Korean include thousands of different characters, and typically require more than one byte to store a given logical character. Characters that occupy two or more bytes of storage are called *multibyte characters*.

Locales

For INFORMIX-SQL (and for Informix database servers and connectivity products), a *locale* is a set of files that specify the linguistic and cultural conventions that the user expects to see when the software runs. A locale can specify these:

- The name of the code set
- The collation order for character-string data
- Culture-specific display formats for other data types
- The correspondence between uppercase and lowercase letters
- Determination of which characters are printable and which are nonprintable

The *Informix Guide to GLS Functionality* provides details of formats for number, currency, and time values. If no locale is specified, then default values are for United States English, which is the **en_us.8859-1** locale on UNIX systems.

INFORMIX-SQL requires the **en_us.0333** locale. It accepts as input any source file containing data values in the format of the client locale.

GLS Features Supported in INFORMIX-SQL

GLS features supported in INFORMIX-SQL include:

- Character data sorting and comparison according to the rules of a national language locale.
- Use of extended-ASCII characters permitted in user-defined names such as database, table, and column names.
- Nationalized money and numeric decimal formats in reports, screen forms, and data assignment statements.
- Character conversion between database data and national language specific keyboards and screens.
- The ability for different users to simultaneously access, on the same server, databases with different locale settings.

Data Types and Menu Options

Figure C-1 and Figure C-2 present an overview of the affected data types and INFORMIX-SQL menu options and keywords. (For information about locale files and locale categories, see the *Informix Guide to GLS Functionality*.)

Figure C-1
Impact of GLS Support on Data Types

Data Type	Impact
CHAR	Transparently maps to NCHAR
VARCHAR	Transparently maps to NVARCHAR
NCHAR	Sorts in the order of the user locale. Available only by way of SQL CREATE TABLE
NVARCHAR	Sorts in the order of the user locale. Available only by way of SQL CREATE TABLE
DECIMAL	Display depends on values in DBFORMAT, DBMONEY, and the NUMERIC category (highest to lowest precedence)
SMALLFLOAT	Display depends on values in DBFORMAT, DBMONEY, and the NUMERIC category (highest to lowest precedence)
FLOAT	Display depends on values in DBFORMAT, DBMONEY, and the NUMERIC category (highest to lowest precedence)
MONEY	Display depends on values in DBFORMAT, DBMONEY, and the MONETARY category (highest to lowest precedence)
DATE	Separator symbol and order of month, day, and year depends on the value in DBDATE. Display of language-specific month and day names depends on installation of message files, whose location is referenced by DBLANG.
DATETIME	Display of language-specific month and day names depends on the installation of message files, whose location is referenced by DBLANG

Figure C-2
Impact of GLS Support on Menu Options and Keywords

Menu Option or Keyword	IMPACT
LOAD	The LOAD statement expects incoming text files to be in the format specified by the GLS and Informix locale settings and environment variables
UNLOAD	Text files produced by an UNLOAD are output in the format specified by GLS and Informix GLS and Informix locale settings and environment variables but without thousands separators
USING	Interpretation of format strings is dependent on settings in DBFORMAT, DBMONEY, DBDATE, and the NUMERIC and MONETARY categories
CREATE TABLE, ALTER TABLE	CHAR and VARCHAR columns defined in non-English locales are created as NCHAR and NVARCHAR. CHAR and VARCHAR columns that behave as CHAR and VARCHAR in these environments can only be created by way of the SQL CREATE TABLE and ALTER TABLE statements
FORMAT	Same as USING except that FORMAT does not support currency symbols
ORDER BY, MATCHES, WHILE, INCLUDE, and IF	Comparisons of character values are based on collation sequences defined by the COLLATE category
LET	Conversions between character and numeric, monetary or date values are dependent on settings in DBFORMAT, DBMONEY, DBDATE, and the NUMERIC and MONETARY categories
UPSHIFT and DOWNSHIFT	Translations between upper and lowercase are specified by the CTYPE category
ASCII (ACE)	Characters generated by particular ASCII values are dependent on which character set is specified by the CTYPE category
DATE	The date displayed contains month and day names specified by the message files pointed to by DBLANG
MENU NAME	Menu names can include locale-specific characters
CALL (to C function)	Called C functions can include locale-specific characters in identifiers, if the C compiler can support these

Date, Time, and Currency Formats

To use localized formats for dates, time, and money values, set the Informix environment variables **DBFORMAT**, **DBMONEY**, and **DBDATE**. Formatting conventions of some East Asian locales require that the **GL_DATE** or **GL_DATETIME** environment variable be set. For more information about these and other environment variables, see the *Informix Guide to GLS Functionality*.

Informix System Error Messages

Informix provides error message translation for a variety of languages. You can use the **DBLANG** environment variable to point to a message directory containing translated messages. Contact your local Informix sales office for a list of available language translations.

The Character Set

INFORMIX-SQL can handle the following non-English characters that are valid in the client locale:

- Names of identifiers
- Values of CHAR and VARCHAR variables and formal arguments
- Characters within TEXT blobs
- Message text, quoted strings, and values returned by functions
- Text within comments, forms, menus, and output from reports

Named entities include variables, functions, cursors, formal arguments, labels, reports, and prepared objects. INFORMIX-SQL has a limit of 50 bytes on the lengths of these names.

The default environment for INFORMIX-SQL is based on the ASCII code set of 128 characters, as listed in Appendix E, "The ASCII Character Set." Each of these encoded values (or *code points*) requires seven bits of a byte to store each of the values 0 through 127, representing the letters, digits, punctuation, and other logical characters of ASCII. Because each ASCII character can be stored within a single byte, ASCII is called a *single-byte* character set. All other character sets that INFORMIX-SQL can support must include ASCII as a subset.

In non-English locales, INFORMIX-SQL can include non-ASCII characters in identifiers if those characters are defined in the code set of the locale that **CLIENT_LOCALE** specifies. In multibyte East Asian locales that support languages whose written form is not alphabet-based, an identifier need not begin with a letter, but the storage length cannot exceed 50 bytes. (A Chinese identifier, for example, that contains 50 logical characters would exceed this limit if any logical character in the identifier required more than one byte of storage.)

You can enter, edit, and display valid characters from the code set of the client locale in INFORMIX-SQL. Whether a given character from a non-English code set is *printable* or *nonprintable* depends on the client locale.

The PERFORM screen transaction processor can process form specifications that include non-English characters that are valid in the client locale. It can also produce compiled forms that can display characters from the client locale, and that can accept such characters in input from the user.

Values that include non-English characters can be passed between INFORMIX-SQL and the database server, if the client and server systems have the same locale. If the locales are different, data can still be transferred between the client and the database server, provided that the client locale includes appropriate code-set conversion tables. See "Configuring the Language Environment" on page C-16 or the *Informix Guide to GLS Functionality*, for information about establishing a locale and about code-set conversion between locales. See also "Handling Code-Set Conversion" on page C-21 of this appendix.

SQL Identifiers

SQL identifiers are the names of database entities, such as table and column names, indexes, and constraints. The first character must be an alphabetic character, as defined by the locale, or an underscore (= ASCII 95) symbol. You can use alphanumeric characters and underscores (_) for the rest of the SQL identifier. Most SQL identifiers can be up to 18 bytes in length. What characters are valid in SQL identifiers depends on the locale of the database server (see "Client Locales and Server Locales" on page C-15). Neither single-byte nor multibyte whitespace characters can appear in SQL identifiers.

SE

For INFORMIX-SE database servers, whether non-English characters are permitted in the names of databases, tables, or log files depends on whether the operating system permits such characters in filenames. ♦

The user interface of INFORMIX-SQL is in English. If edit fields contain multibyte characters, there is no checking, and the results might be unpredictable. SQL statements can include valid non-English identifiers for some database entities.The following tables summarize the instances where non-English characters are valid as identifiers.

SQL Identifier	Allow Non-English Characters?
Column name	Yes
Constraint name	Yes
Database name	Yes (Operating System limitations on INFORMIX-SE)
Index name	Yes
Log filename	Yes (Operating System limitations on INFORMIX-SE)
Stored procedure name	Yes
Synonym	Yes
Table name	Yes (Operating System limitations on INFORMIX-SE)
View name	Yes

Input and output filenames cannot be localized. Only ASCII characters are valid in input and output pathnames or filenames.

Collation Sequence

The collation (sorting) sequence is implied by the *code-set order* in the files that define the client locale. (Any collating that is specified by the **COLLATE** category of the client locale is ignored.) Collation in SQL operations (where the database server uses its own collation sequence) depends on the data type and on the server locale (which can specify a localized order of collation). It is possible for INFORMIX-SQL and the database server to use a different collating sequence, or for INFORMIX-SQL to connect to two or more servers that use different collating sequences. The collation sequence can affect the value of Boolean expressions that use relational operators, and the sorted order of rows in queries and in reports.

East Asian Language Support

INFORMIX-SQL can accept Asian languages that use multibyte code sets. The following features are supported in multibyte locales:

- Menu items, identifiers, and text labels in the native language
- Features to avoid the creation of partial characters
- Non-English data
- Cultural conventions, including the representation of date, time, currency, and numeric values, and localized collation
- Kinsoku processing for Japanese language text with WORDWRAP
- Text geometry that adjusts automatically to meet localization needs
- Comparisons that adopt the comparison rules and collating sequence that the locale defines implicitly (SQL comparison and collation depend on the database server.)

This version of INFORMIX-SQL does not support composite characters, such as are required in some code sets that support the Thai language.

Character string values can include multibyte characters that are supported by the client locale in contexts like these:

- Character expressions and multiple-value character expressions
- Literal values within quoted strings
- Variables, formal arguments, and returned values of CHAR, VARCHAR, and TEXT data types

Multibyte characters can also appear in user-defined query criteria that specify the SQL identifier of any of the database objects listed in the table on "SQL Identifiers" on page C-8. INFORMIX-SQL does not, however, support multibyte characters as currency symbols or as separators in display formats specified by the **DBDATE** or **DBFORMAT** environment variables.

Logical Characters

Within a single-byte locale, every character of data within character-string values requires only a single byte of memory storage, and a single character position for display by a character-mode device.

This simple one-to-one relationship in character-string operations between data characters, display width, and storage requirements does not exist in East Asian locales that support multibyte characters. In such locales, a single logical character might correspond to a single byte or to two or more bytes. In such locales, it becomes necessary to distinguish among the *logical characters* within a string, the *display width* that the corresponding glyph occupies in a display or in report output, and the number of *bytes* of memory storage that must be allocated to hold the string.

In locales that support multibyte characters, some built-in functions and operators that process string values operate on logical characters, rather than on bytes. For code sets that use multibyte characters, this modifies the byte-based behavior of several features in INFORMIX-SQL 6.x (and earlier) releases. A single logical character can occupy one or more character positions in a screen display or in output of a report, and requires at least one byte of storage, and possibly more than one.

Declaring the CHAR or VARCHAR data types of variables, formal arguments, and returned values is *byte*-based. Runtime processing of some character strings, however, is done on a *logical character* basis in multibyte locales.

Partial Characters

The most important motivation for distinguishing between logical characters and their component bytes is the need to avoid partial characters. These are fragments of multibyte characters. Entering partial characters into the database implies corruption of the database, and risks malfunction of the database server.

Partial characters are created when a multibyte character is truncated or split up in such a manner that the original sequence of bytes is not retained. Partial characters can be created during operations like the following:

- Substring operations
- INSERT and UPDATE operations of SQL

- Word wrapping in reports and screen displays
- Buffer to buffer copy

INFORMIX-SQL does not allow partial characters and handles them as follows:

- Replaces truncated multibyte characters by single-byte whitespaces
- Wraps words in a way that ensures that no partial characters are created in reports and screen displays
- Performs code-set conversion in a way that ensures that no partial characters are created

For example, suppose that the following SELECT statement of SQL:

```
SELECT col1[3,5] FROM tab1
```

retrieved three data values from **col1** (where **col1** is a CHAR, NCHAR, NVARCHAR, or VARCHAR column); here the first line is not a data value but indicates the alignment of bytes within the substrings:

```
AA²BB²AA       becomes      "s¹Bs¹"
ABA²C²AA       becomes      "A²s¹"
A²B²CABC      becomes      "B²C"
```

Here the notation s^1 denotes a single-byte whitespace. Any uppercase letter followed by a superscript (^{2}) means an East Asian character with multibyte storage width; for simplicity, this example assumes a 2-byte storage requirement for the multibyte characters. In the first example, the A^2 would become a partial character in the substring, so it is replaced by a single-byte whitespace. In the same substring, the B^2 would lose its trailing byte, so a similar replacement takes place.

Installing INFORMIX-SQL in Non-English Locales

This section identifies the general requirements for installation of INFORMIX-SQL in non-English locales. Because "non-English" refers to all locales other than **en_us.8859-1**, most locales of the English-speaking world are "non-English" in this context, as are the locales of most of the rest of the world.

The directory structure of Informix GLS products is shown in Figure C-3.

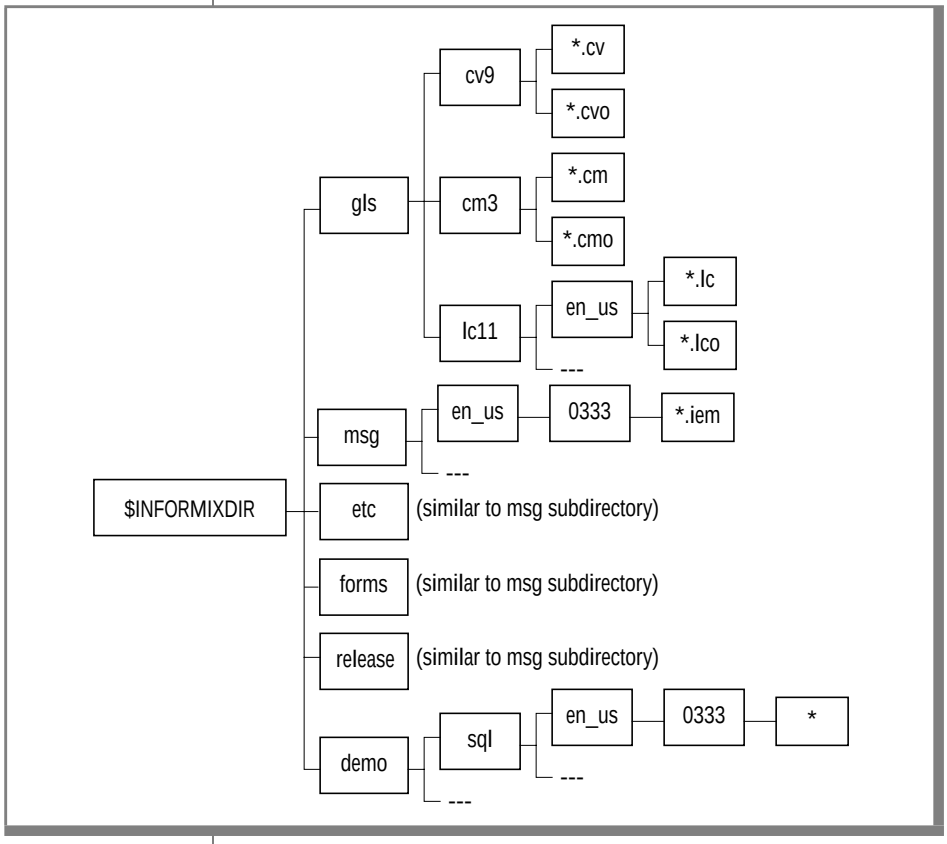


Figure C-3
Directory Structure of GLS Products

Language Supplements

Use of INFORMIX-SQL with some non-English languages might require an Informix *language supplement* specific to the conventions of the country or language. Language supplements are currently required, for example, for Informix database servers to support each of the following East Asian languages.

Country or Language	Informix Language Supplement
People's Republic of China	Language Supplement ZHCN 7.20
Taiwanese	Language Supplement ZHTW 7.20
Japanese	Language Supplement JA 7.20
Korean	Language Supplement KO 7.20
Thai (simplified)	Language Supplement TH 7.20

Language supplements for these East Asian languages include locale files, translated message files, and translated menu files. Localized versions of INFORMIX-SQL for East Asian locales (for example, Japanese INFORMIX-SQL) will include the relevant files. See the release notes for additional information.

A corresponding International Language Supplement includes locale files and code-set conversion files for most European languages. Because most of these files are included with the Client SDK software that is provided with INFORMIX-SQL, this supplement need not be purchased by INFORMIX-SQL customers unless the required locale is not included with INFORMIX-SQL.

When the Informix database server is installed in locales based on non-English European languages, both the default (English) database server and the International Language Supplement must be installed.

When INFORMIX-SQL is installed, the locale files must also be installed. Contact your Informix sales office for information regarding current support for specific locales.

Locales that INFORMIX-SQL Supports

A *locale* is the part of the processing environment that defines conventions for a given language or culture, such as formatting time and money values, and classifying, converting, and collating characters. The Informix GLS locale definition is similar to the X/Open CAE Specification.

Languages that INFORMIX-SQL supports include the following:

- People's Republic of China
- Taiwanese
- Japanese
- Korean
- Eastern European (Latin)
- Eastern European (Cyrillic)
- Western European (Latin)
- Greek
- Turkish

"Latin" denotes what is also called the "Roman" alphabet in U.S. English. In any locale, INFORMIX-SQL requires at least one font that supports the code set.

INFORMIX-SQL provides limited support for the Thai language with Language Supplement TH 7.20, for non-composite Thai characters. (INFORMIX-SQL does not support composite Thai characters.)

Client Locales and Server Locales

The locale of the system on which INFORMIX-SQL is running is called the *client locale*. The locale of the database server is called the *server locale*. "Handling Code-Set Conversion" on page C-21 describes special procedures that might be required if the client locale and the server locale are not identical.

Specifying Locales

INFORMIX-SQL requires that locales be specified, through environment variables and settings in locale files, on UNIX systems that support the database server.

For details about configuring UNIX systems for global language support and for an example of non-English locale files, see the *Informix Guide to GLS Functionality*. For additional information about setting environment variables, see also "Configuring the Language Environment" on page C-16.

Configuring the Language Environment

Using the GLS features of INFORMIX-SQL with Informix database servers involves several compatibility issues:

- The English servers create English databases with ASCII data. For these, INFORMIX-SQL must access the servers with **DB_LOCALE** set to **en_us.8859-1**.
- The 5.x ALS versions of Informix servers can use variables such as **DBCODESET** and **DBCSOVERRIDE** as substitutes for **DB_LOCALE** and **DBCONNECT**, respectively.
- The 5.xALS versions use **DBASCIIBC** to emulate the 4.x ASCII servers. This environment variable should be set if such behavior is desired.
- The **SERVER_LOCALE** environment variable is set on the database server, not on the INFORMIX-SQL client. This specifies the locale that the database server uses to read or write operating system files. If this is not set, the default is U.S. English (**en_us.8859-1**).

If no setting is specified, INFORMIX-SQL uses an English locale.

The non-internationalized portions of the product are initialized with the default (U.S. English) locale. That is, both **CLIENT_LOCALE** and **DB_LOCALE** (**en_us.8859-1**) are set to English. This initialization is necessary because many common functions are shared between the internationalized and non-internationalized components.

Important: *Except for **DBFORMAT**, all the environment variables that are described in the sections that follow apply to Informix database servers.*

The following considerations apply:

- INFORMIX-SQL cannot support connections to different databases with different locales concurrently; for example, in extended joins.
- The environment variables discussed here deal with the environment **DB_LOCALE** that is passed to the server.
- **CLIENT_LOCALE** cannot be changed dynamically during execution.
- The previous point has one exception: the **CLIENT_LOCALE** can always be set to English (because English is a subset of all locales).

When connecting to a GLS, NLS, or ALS (Asian Language Support) database, the **DB_LOCALE** code set should match the **DB_LOCALE** code set of the database. Otherwise, data corruption can occur, because no validation of code-set compatibility is performed by the server. An ALS server can refuse the connection when the code sets do not match, but an NLS server cannot.

Environment Variables That Support GLS

This section examines the environment variables that support the GLS capabilities of INFORMIX-SQL, including the following environment variables:

- **DBDATE** defines date display formats.
- **DBMONEY** defines monetary display formats.
- **DBFORMAT** defines numeric and monetary display formats and has more options than **DBMONEY**.

INFORMIX-SQL also supports the following GLS environment variables:

- **DB_APICODE** specifies a code set that has a mapping file.
- **DB_LOCALE** is the locale of the database to which INFORMIX-SQL is connected.
- **CLIENT_LOCALE** is the locale of the system that is executing INFORMIX-SQL.
- **DBLANG** points to the directory for Informix error messages.
- **GL_DATE** defines date displays, including East Asian formats.
- **GL_DATETIME** defines date and time displays, including East Asian formats.

- **SERVER_LOCALE** is the locale of the database server for file I/O.

INFORMIX-SQL does not use **DB_LOCALE** directly; this variable, as well as **DBLANG**, is used by the GLS version of Client SDK. See the *Informix Guide to GLS Functionality* for details on how **DBLANG, DB_LOCALE, GL_DATE** and **GL_DATETIME** are set. For details of other Informix environment variables, see Appendix B, "Setting Environment Variables."

DBAPICODE

This environment variable specifies the name of a mapping file for peripheral devices (for example, a keyboard, a display terminal, or a printer) whose character set is different from that of the database server.

DB_LOCALE

This environment variable specifies the locale of the database to which INFORMIX-SQL is connected. The format for setting **DB_LOCALE** is DB_LOCALE=<*locale*>.

The following points should be noted regarding **DB_LOCALE**:

- The locale of the database must match the value specified in **DB_LOCALE**. If it does not match, the database connection might be refused (unless **DBCSOVERRIDE** is set to 1), depending on the server version.
- If a database is created, then this new database has the value specified by **DB_LOCALE**.
- If **DB_LOCALE** is invalid, either because of wrong formatting or specifying a locale that does not exist, then an error is issued.
- If the code set implied by **DB_LOCALE** cannot be converted to what **CLIENT_LOCALE** implies, or vice versa, an error is issued.
- If **DB_LOCALE** is not specified, there is no default value; in this case, the GLS version of Client SDK behaves as if code-set conversion were not needed.

CLIENT_LOCALE

This environment variable specifies the locale of the (input) source code and the compiled code (to be generated). This is also the locale of the error files (if any) and the intermediate files. The format of **CLIENT_LOCALE** is the same as that of **DB_LOCALE**:

- The characters that reach the user interface (the non-ASCII characters) must be in the **CLIENT_LOCALE**.
- If **DB_LOCALE** is invalid, either because of incorrect formatting or specifying a locale that does not exist, an error is issued.
- The **DB_LOCALE** and **CLIENT_LOCALE** settings need to be compatible, meaning there should be proper code-set conversion tables between them. Otherwise, an error is generated.
- Collation follows the code-set order of **CLIENT_LOCALE**, except in SQL statements (where the database server uses its own collation sequence). Any **COLLATE** specification is ignored.

DBLANG

The value of **DBLANG** is used to complete the pathname to the directories that contain the required message, help, and demo files. The format of **DBLANG** is the same as that of **DB_LOCALE**.

If **DBLANG** is not set, the value defaults to that of **CLIENT_LOCALE**.

See also the description of **DBLANG** in the *Informix Guide to GLS Functionality*.

DBDATE

The **DBDATE** environment variable has been modified to support era-based dates (Japanese and Taiwanese). The days of the week and months of the year (in local form) are stored in the locale files. If this environment variable is set, it might override other means of specifying date formats.

DBMONEY

This environment variable has been modified to accept multibyte currency symbols. INFORMIX-SQL must read the value of **DBMONEY** (or **DBFORMAT**) and be able to correctly process multibyte characters as currency symbols. If **DBMONEY** is set, its value might override other means of specifying currency formats.

DBFORMAT

This environment variable has been modified to accept multibyte currency symbols. Unlike the version of **DBFORMAT** for English products, display of the decimal point is optional, rather than mandatory, in INFORMIX-SQL.

If **DBFORMAT** is set, its value can override other means of specifying number or monetary formats.

See also the descriptions of **DBDATE**, **DBFORMAT**, and **DBMONEY** in Appendix B and the *Informix Guide to SQL: Reference*.

The **glfiles** utility is described in the *Informix Guide to GLS Functionality*. This utility allows you to generate lists of the following files:

- GLS locales available in the system
- Informix code-set conversion files available
- Informix code-set files available

Default Values of GLS Environment Settings

Default values assumed by INFORMIX-SQL (which differ from those of ALS environments) are described in this section.

The following table shows the values assumed by INFORMIX-SQL when you define only some of the required values of locales.

(A value of ja-jp.ujis is assumed in the following example, CL means **CLIENT_LOCALE**, and DL means **DB_LOCALE**.)

User Defined				Values in Product	
CL Defined	**CL Value**	**DL Defined**	**DL Value**	**CL Value**	**DL Value**
No	--	No	--	en_us.8859	en_us.8859
Yes	ja_jp.ujis	No	--	ja_jp.ujis	ja_jp.ujis
Yes	ja_jp.ujis	Yes	ja_jp.ujis	ja_jp.ujis	ja_jp.ujis
No	--	Yes	ja_jp.ujis	en_us.8859	ja_jp.ujis

If you do not set the **DBLANG** environment variable, it is set to the value of **CLIENT_LOCALE**.

System Environment Variables

The value of the X/Open-defined **LANG** environment variable specifies the language environment. There is no standardization of **LANG** locale values between systems. Exact values to specify for locale variables are specific to the system and also depend on which language supplements have been installed on the system.

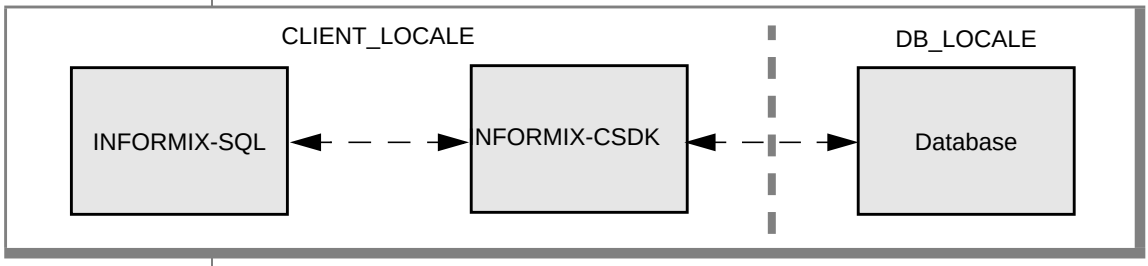
For more information about the **LANG** environment variable, see the *Informix Guide to SQL: Reference*.

Handling Code-Set Conversion

The process of converting characters at the locale where INFORMIX-SQL is running to characters at the locale of the database server (or vice versa) is called *code-set conversion*. If INFORMIX-SQL needs to run on computers that encode different character sets, it might be necessary to enable code-set conversion. This section provides some background and details.

Code-set conversion is performed by Client SDK; no explicit code-set conversion is done by INFORMIX-SQL. Figure C-4 shows the relationship between INFORMIX-SQL, Client SDK, and the database.

Figure C-4
Processes and Their Locales



The code sets in the **CLIENT_LOCALE** can differ from those in **DB_LOCALE**.

Code-set conversion is done by way of a code-set conversion file. Files for code-set conversion between **CLIENT_LOCALE** and **DB_LOCALE** need to be present on the client. For conversion to take place, conversion files need to be present in the **$INFORMIXDIR/gls/cv** directory.

For details of converting between client and server code sets, see the sections that follow. See also the *Informix Guide to GLS Functionality*.

What Is Code-Set Conversion?

Different operating systems sometimes encode the same characters in different ways. For example, the character *a-circumflex* is encoded:

- in Windows code page 1252 as hexadecimal 0xE2.
- in IBM CCSID 437 as hexadecimal 0x83.

If the encoding for *a-circumflex* on the Windows system is sent unchanged to the IBM system, it will be printed as the Greek character gamma. This happens because, on the IBM system, gamma is encoded as 0xE2.

This means character data strings that are passed between two computers using different character set encodings must be converted between the two different encodings. Otherwise, character data originating from one computer will not be correctly displayed or processed on the other computer.

This appendix uses the term *code set* in the same way that the Windows documentation uses the terms *character set* and *code page*.

Converting character data from one encoding schema to another is called *code-set conversion*. If a code-set conversion is required from computer A to computer B, it is also required from computer B to computer A. You must explicitly enable code-set conversion; no conversion is done by default.

What Code-Set Conversion Is Not

Code-set conversion is not a semantic translation; that is, it does not convert words between different languages. For example, it does not convert between English *yes* and French *oui*. It only ensures that each character is processed and printed the same, regardless of how the characters are encoded.

Code-set conversion does not create a character in the target code set if the character exists only in the source code set. For example, if the character *a-circumflex* is being passed to a computer whose code set does not contain an *a-circumflex* character, the target computer will never be able to exactly process or print the *a-circumflex* character. This situation is described in more detail in “Mismatch Processing” on page C-24.

When You Do Not Need Code-Set Conversion

You do not need code-set conversion in any of the following situations:

- The client and the server are on the same computer.
- The code set of your client and of all the databases to which you are connecting are the same.
- The subset of characters that you will be sending between the client and the server are encoded identically. For example, if you are sending only English characters between a client and a server, and each English character has the same encoding on both computers, no code-set conversion is required. In this case, the non-English characters can have different encodings.
- The character-string data values are passed from the client to the server for storage only and are neither processed nor printed by the server. For example, no code-set conversion is required if a client:
  - passes character-string data to the server.
  - does not process or print the data on the server computer.
  - retrieves the same data for processing or printing on computers that use the same code set as the client that populated the database.

Sorting data by using the ORDER BY statement or retrieving data by using a LIKE or MATCHES clause, however, will probably produce erroneous results if the data strings are not converted before they are stored.

What Data Values Are Converted

If you enable code-set conversion, data values are converted by Client SDK from the INFORMIX-SQL client to the database server, and from the server to the client. The CHAR, VARCHAR, and TEXT blob data types are converted, as are column names, table names, database names, and SQL command text.

Mismatch Processing

If both code sets encode exactly the same characters, then mismatch handling is unnecessary. If the source code set contains any characters that are not contained in the target code set, however, the conversion must define how the mismatched characters are to be mapped to the target code set.

Four ways code-set conversions handle mismatch processing are as follows:

- **Round-trip conversion.** This maps each mismatched character in the source code set to a unique character in the target code set. On the return, the original character is mapped back to itself. This guarantees that a two-way conversion will result in no loss of information; however, data converted in only one direction might confuse the processing or printing on the target computer.
- **Substitution conversion.** This maps all mismatched characters in the source code set to a single specific character in the target code set that serves to highlight mismatched characters. This guarantees that a one-way conversion will clearly show the mismatched characters; however, a two-way conversion will result in information loss if mismatched characters are transferred.
- **Graphical replacement conversion.** This maps each mismatched character in the source code set to a character in the target code set that resembles the source character (this includes mapping one-character ligatures to their two-character equivalents). This might confuse printing on the target computer. Round-trip conversions should contain as many graphical replacement conversions as possible.
- **Substitution plus graphical replacement.** This maps as many mismatched characters as possible to their graphical replacements, and maps the remaining mismatched characters to the substitution character.

Informix-supplied code-set conversion source files have header comments that indicate which method was used.

Enabling Code-Set Conversion

Code-set conversion on UNIX is handled by UNIX environment variables.

To establish code-set conversion

1. Determine the code set used by the client.
2. Determine the code set used by all the databases to which this client will be connecting in a single connection.
3. Specify the conversion filenames.

Because each operating system has its own way of declaring the code set it is using, consult your UNIX operating system documentation or your system administrator to determine the code set used by the client computer.

Your system administrator should also know which code set is being used by the database.

Set the **DBAPICODE** environment variable to specify a code set that has a mapping file in the message directory **$INFORMIXDIR/msg** (or a directory pointed to by the **LANG** or **DBLANG** value). The Informix **crtcmap** utility helps you to create mapping files.

For detailed information about **DBAPICODE** and the **crtcmap** utility, see the *Informix Guide to SQL: Reference.* ♦

Appendix

D

Modifying termcap and terminfo

You can include color and graphics characters in your PERFORM screen forms. The meaning of these characters, however, is terminal dependent. To determine terminal-dependent characteristics, INFORMIX-SQL uses the information in the **termcap** file or in the **terminfo** directory. INFORMIX-SQL uses the **INFORMIXTERM** environment variable to determine whether to use **termcap** or **terminfo**. For more information about **INFORMIXTERM**, read the discussion of environment variables in the 'Environment Variables' appendix or in the Preface.

With INFORMIX-SQL, Informix distributes **termcap** files that contain additional capabilities for many common terminals (such as the Wyse 50 and the Televideo 950). This appendix describes these capabilities, as well as the general format of **termcap** and **terminfo** entries.

Because **terminfo** does not support color, you can only use INFORMIX-SQL color functionality with **termcap**. If you want to use color in INFORMIX-SQL screen forms, you must set **INFORMIXTERM** to **termcap.**

You can use the information in this appendix, combined with the information in your terminal manual, to modify the contents of your **termcap** file or **terminfo** files. This appendix is divided into two main sections, **termcap** and **terminfo**. Depending on which you are using, you should read the appropriate section.

termcap

When INFORMIX-SQL is installed on your system, a **termcap** file is placed in the **etc** subdirectory of **$INFORMIXDIR**. This file is a superset of an operating-system **termcap** file. The Informix **termcap** file contains additional capabilities for many terminals. You might want to modify this file further in the following instances:

- The entry for your terminal has not been modified to include color-change and intensity-change capabilities.
- You want to specify or alter the graphics characters used for borders.

Tip: *Some terminals cannot support color or graphics characters. You should read this appendix and the user guide that comes with your terminal to determine whether or not the changes described in this appendix are applicable to your terminal.*

Format of a *termcap* Entry

This section describes the general format of **termcap** entries. For a complete description of **termcap**, refer to your operating-system documentation.

A **termcap** entry contains a list of names for the terminal, followed by a list of the terminal's capabilities. The three types of capabilities are:

- Boolean
- Numeric
- String

All **termcap** entries have the following format:

- ESCAPE is specified as a backslash (\ ) followed by the letter E, and CONTROL is specified as a caret (^). Do not use the ESCAPE or CONTROL keys to indicate escape sequences or control characters in a **termcap** entry.
- Each capability, including the last one in the entry, is followed by a colon (:).
- Entries must be defined on a single logical line; a backslash (\ ) appears at the end of each line that wraps to the next line.

Figure D-1 shows a basic **termcap** entry for the Wyse 50 terminal.

```
# Entry for Wyse 50:

w5|wy50|wyse50:\
    :if=/usr/lib/tabset/std:\
    :al=\EE:am:bs:ce=\Et:cm=\E=%+ %+ :cl=\E*:co#80:\
    :dc=\EW:dl=\ER:ho=^^:ei=:kh=^^:im=:ic=\EQ:in:li#24:\
    :nd=^L:pt:se=\EG0:so=\EG4:sg#1:ug#1:\
    :up=^K:ku=^K:kd=^J:kl=^H:kr=^L:kb=:\
    :k0=^A@^M:k1=^AA^M:k2=^AB^M:k3=^AC^M:k4=^AD^M:\
    :k5=^AE^M:k6=^AF^M:k7=^AG^M:\
    :HI=^|:Po=^R:Pe=^T:
```

Figure D-1 *Wyse 50 termcap Entry*

Tip: *Comment lines begin with a pound sign (#).*

Terminal Names

A **termcap** entry starts with one or more names for the terminal, each of which is separated by a vertical bar (|). For example, the **termcap** entry for the Wyse 50 terminal starts with the following line:

```
w5|wy50|wyse50:\
```

The **termcap** entry can be accessed with any one of these names.

Boolean Capabilities

A Boolean capability is a two-character code that indicates whether or not a terminal has a specific feature. If the Boolean capability is present in the **termcap** entry, the terminal has that particular feature. Figure D-2 shows some of the Boolean capabilities for the Wyse 50 terminal.

```
    bs:am:

    #   bs   backspace with CTRL-H
    #   am   automatic margins
```

Figure D-2 *Boolean Capabilities for the Wyse 50*

Numeric Capabilities

A numeric capability is a two-character code followed by a pound symbol (#) and a value. Figure D-3 shows the numeric capabilities for the number of columns and the number of lines on a Wyse 50 terminal.

```
:co#80:li#24:

#   co   number of columns in a line
#   li   number of lines on the screen
```

Figure D-3
Numeric Capabilities for the Wyse 50

Similarly, **sg** is a numeric capability that indicates the number of character positions required on the screen for reverse video. The entry `:sg#1:` indicates that a terminal requires one additional character position when reverse video is turned on or off. If you do not include a particular numeric capability, INFORMIX-SQL assumes that the value is zero.

String Capabilities

A string capability specifies a sequence that can be used to perform a terminal operation. A string capability is a two-character code followed by an equal sign (=) and a string ending at the next delimiter (:).

Most **termcap** entries include string capabilities for clearing the screen, cursor movement, arrow keys, the underscore, function keys, and so on. Figure D-4 shows many of the string capabilities for the Wyse 50 terminal.

```
:ce=\Et:cl=\E*:\
:nd=^L:up=^K:\
:so=\EG4:se=\EG0:\
:ku=^K:kd=^J:kr=^L:kl=^H:\
:k0=^A@^M:k1=^AA^M:k2=^AB^M:k3=^AC^M:

#   ce=\Etclear to end of line
#   cl=\E*clear the screen
#   nd=^Lnon-destructive cursor right
#   up=^Kup one line
#
#   so=\EG4start stand-out
#   se=\EG0end stand-out
#
#   ku=^Kup arrow key
#   kd=^Jdown arrow key
#   kr=^Lright arrow key
#   kl=^Hleft arrow key
#
#   k0=^A@^Mfunction key F1
#   k1=^AA^Mfunction key F2
#   k2=^AB^Mfunction key F3
#   k3=^AC^Mfunction key F4
```

Figure D-4
String Capabilities for the Wyse 50

Specifying Graphics Characters in Screen Forms

INFORMIX-SQL uses characters defined in the **termcap** file to draw the borders of boxes and other rectangular shapes that appear in a screen form. If no characters are defined in the **termcap** file, INFORMIX-SQL uses the hyphen (–) for horizontal lines, the vertical bar (|) for vertical lines, and the plus sign (+) for corners.

The **termcap** file provided with INFORMIX-SQL contains border character definitions for many common terminals. You can look at the **termcap** file to see if the entry for your terminal has been modified to include these definitions. If your terminal entry does not contain border character definitions, or if you want to specify alternative border characters, you or your system administrator can modify the **termcap** file.

Perform the following steps to modify the definition for your terminal type in the **termcap** file:

1. Determine the escape sequences for turning graphics mode on and off. This information is located in the manual that comes with your terminal. For example, on Wyse 50 terminals, the escape sequence for entering graphics mode is ESC H^B and the escape sequence for leaving graphics mode is ESC H^C.

Tip: *Terminals without a graphics mode do not have this escape sequence. The procedure for specifying alternative border characters on a non-graphics terminal is discussed at the end of this section.*

2. Identify the ASCII equivalents for the six graphics characters that INFORMIX-SQL requires to draw the border. (The ASCII equivalent of a graphics character is the key you would press in graphics mode to obtain the indicated character.)

   Figure D-5 shows the graphics characters and the ASCII equivalents for a Wyse 50 terminal.

   Figure D-5
   Wyse 50 ASCII Equivalents for Border Graphics Characters

Window Border Position	Graphics Character	ASCII Equivalent
upper left corner	⌜	2
lower left corner	⌞	1
upper right corner	⌝	3
lower right corner	⌟	5
horizontal	-	z
vertical	|	6

   Again, this information should be located in the manual that comes with your terminal.

3. Edit the **termcap** entry for your terminal.

Tip: *You might want to make a copy of your* ***termcap*** *file before you edit it. You can use the TERMCAP environment variable to point to whichever copy of the* ***termcap*** *file you want to access.*

Use the format

```
termcap-capability=value
```

to enter values for the following **termcap** capabilities:

gs The escape sequence for entering graphics mode. In the termcap file, ESCAPE is represented as a backslash (\ ) followed by the letter E; CONTROL is represented as a caret (^). For example, the Wyse 50 escape sequence ESCAPE-H CONTROL-B is represented as \EH^B.

ge The escape sequence for leaving graphics mode. For example, the Wyse 50 escape sequence ESCAPE-H CONTROL-C is represented as \EH^C.

gb The concatenated, ordered list of ASCII equivalents for the six graphics characters used to draw the border. Use the following order: upper left corner, lower left corner, upper right corner, lower right corner, horizontal lines, vertical lines.

Follow these guidelines when you insert information in the **termcap** entry:

- Delimit entries with a colon (:).
- End each continuing line with a backslash (\ ).
- End the last line in the entry with a colon.

For example, if you are using a Wyse 50 terminal, you would add the following information in the **termcap** entry for the Wyse 50:

```
    :gs=\EH^B:\
# sets gs to ESC H CTRL B
    :ge=\EH^C:\
# sets ge to ESC H CTRL C
    :gb=2135z6:\
# sets gb to the ASCII equivalents
# of graphics characters for upper
# left, lower left, upper right,
# lower right, horizontal,
# and vertical
```

If you prefer, you can enter this information in a linear sequence:

```
:gs=\EH^B:ge=\EH^C:gb=2135z6:\
```

Terminals Without Graphics Capabilities

For terminals without graphics capabilities, you must enter a blank value for the **gs** and **ge** capabilities. For **gb**, enter the characters you want INFORMIX-SQL to use for the window border.

The following example shows possible values for **gs**, **ge**, and **gb** in an entry for a terminal without graphics capabilities. In this example, window borders would be drawn using underscores (_) for horizontal lines, vertical bars (|) for vertical lines, periods (.) for the top corners, and vertical bars (|) for the lower corners.

```
:gs=:ge=:gb=.|.|_|:
```

INFORMIX-SQL uses the graphics characters in the **termcap** file when you specify a screen border in a PERFORM screen.

Adding Color and Intensity

Many of the terminal entries in the Informix **termcap** file (in the **etc** subdirectory of **$INFORMIXDIR**) have been modified to include color or intensity capabilities or both. You can view the **termcap** file to determine if the entry for your terminal type includes these capabilities. If your terminal entry includes the **ZA** capability, your terminal is set up for color or intensity or both. If it does not, you can add color and intensity capabilities by using the information in this section. The following topics are outlined in this section:

- Color and intensity
- The **ZA** capability
- Stack operations
- Examples

Read these topics before you modify your terminal entry.

Color and Intensity Attributes

You can display your PERFORM screen on either a monochrome or a color terminal. If you set up the **termcap** files as described here, color attributes and intensity attributes are related, as shown in Figure D-6.

Figure D-6
Color-Monochrome Correspondence

Number	Color Terminal	Monochrome Terminal
0	WHITE	NORMAL
1	YELLOW	BOLD
2	MAGENTA	BOLD
3	RED	BOLD†
4	CYAN	DIM
5	GREEN	DIM
6	BLUE	DIM†
7	BLACK	INVISIBLE

The background for colors is BLACK in all cases. In Figure D-6, the † signifies that, if the keyword BOLD is indicated as the attribute, the field will be RED on a color terminal, or if the keyword DIM is indicated as the attribute, the field will be BLUE on a color terminal.

In either color or monochrome mode, you can add the REVERSE, BLINK, or UNDERLINE attributes if your terminal supports them. You can select only one of these three attributes.

The ZA String Capability

INFORMIX-SQL uses a parameterized string capability **ZA** in the **termcap** file to determine color assignments. Unlike other **termcap** string capabilities that you set equal to a literal sequence of ASCII characters, **ZA** is a function string that depends on four parameters:

Parameter 1 (p1) Color number between 0 and 7 (see Figure D-6)

Parameter 2 (p2) 0 = Normal; 1 = Reverse

Parameter 3 (p3) 0 = No-Blink; 1 = Blink

Parameter 4 (p4) 0 = No-Underscore; 1 = Underscore

ZA uses the values of these four parameters and a stack machine to determine which characters to send to the terminal. The **ZA** function is called and these parameters are evaluated when a color attribute specification is encountered during PERFORM. You can use the information in your terminal manual to set the **ZA** parameters to the correct values for your terminal.

To define the **ZA** string for your terminal, you use *stack operators* to push and pop values onto and off the *stack*. The next section describes several stack operators. Use these descriptions and the subsequent examples to understand how to define the string for your terminal.

Stack Operations

The **ZA** string uses stack operations to either push values onto the stack or pop values off the stack. Typically, the instructions in the **ZA** string push a parameter onto the stack, compare it to one or more constants, and then send an appropriate sequence of characters to the terminal. More complex operations are often necessary and, by storing the display attributes in static stack machine registers (named **a** through **z**), you can achieve terminal-specific optimizations.

A summary follows of the different stack operators you can use to write the descriptions. For a complete discussion of stack operators, consult your operating system documentation.

Operators that Send Characters to the Terminal

%d pops a numeric value from the stack and sends a maximum of three digits to the terminal. For example, if the value 145 is at the top of the stack, %d pops the value off the stack and sends the ASCII representations of 1, 4, and 5 to the terminal. If the value 2005 is at the top of the stack, %d pops the value off the stack and sends the ASCII representation of 5 to the terminal.

%2d pops a numeric value from the stack and sends a maximum of two digits to the terminal, padding to two places. For example, if the value 145 is at the top of the stack, %2d pops the value off the stack and sends the ASCII representations of 4 and 5 to the terminal. If the value 5 is at the top of the stack, %2d pops the value off the stack and sends the ASCII representations of 0 and 5 to the terminal.

%3d pops a numeric value from the stack and sends a maximum of three digits to the terminal, padding to three places. For example, if the value 7 is at the top of the stack, %3d pops the value off the stack and sends the ASCII representations of 0, 0, and 7 to the terminal.

%c pops a single character from the stack and sends it to the terminal.

Operators that Manipulate the Stack

%p[1-9] pushes the value of the specified parameter on the stack. The notation for parameters is p1, p2, ... p9. For example, if the value of p1 is 3, %p1 pushes 3 on the stack.

%P[a-z] pops a value from the stack and stores it in the specified variable. The notation for variables is Pa, Pb, ... Pz. For example, if the value 45 is on the top of the stack, %Pb pops 45 from the stack and stores it in the variable Pb.

%g[a-z] gets the value stored in the corresponding variable (P[a-z]) and pushes it on the stack. For example, if the value 45 is stored in the variable Pb, %gb gets 45 from Pb and pushes it on the stack.

%´c´ pushes a single character on the stack. For example, %'k' pushes k on the stack.

%{n} pushes an integer constant on the stack. The integer can be any length and can be either positive or negative. For example, %{0} pushes the value 0 on the stack.

%S[a-z] pops a value from the stack and stores it in the specified static variable. (Static storage is nonvolatile since the stored value remains from one attribute evaluation to the next.) The notation for static variables is Sa, Sb, ... Sz. For example, if the value 45 is on the top of the stack, %Sb pops 45 from the stack and stores it in the static variable Sb. This value is accessible for the duration of the INFORMIX-SQL program.

%G[a-z] gets the value stored in the corresponding static variable (S[a-z]) and pushes it on the stack. For example, if the value 45 is stored in the variable Sb, %Gb gets 45 from Sb and pushes it on the stack.

Arithmetic Operators

Each arithmetic operator pops the top two values from the stack, performs an operation, and pushes the result on the stack.

%+ Addition. For example, %{2}%{3}%+ is equivalent to 2+3.

%- Subtraction. For example, %{7}%{3}%- is equivalent to 7-3.

%* Multiplication. For example, %{6}%{3}%* is equivalent to 6*3.

%/ Integer division. For example, %{7}%{3}%/ is equivalent to 7/3 and produces a result of 2.

%m Modulus (or remainder). For example, %{7}%{3}%m is equivalent to (7 mod 3) and produces a result of 1.

Bit Operators

The following bit operators pop the top two values from the stack, perform an operation, and push the result on the stack:

%& Bit-and. For example, `%{12}%{21}%&` is equivalent to (12 and 21) and produces a result of 4.

```
        Binary              Decimal
  0   1   1   0   0     =    12
  1   0   1   0   1     =    21
  ----------------------     and
  0   0   1   0   0     =    4
```

%| Bit-or. For example, `%{12}%{21}%|` is equivalent to (12 or 21) and produces a result of 29.

```
        Binary              Decimal
  0   1   1   0   0     =    12
  1   0   1   0   1     =    21
  ----------------------     or
  1   1   1   0   1     =    29
```

%^ Exclusive-or. For example, `%{12}%{21}%^` is equivalent to (12 exclusive-or 21) and produces a result of 25.

```
        Binary              Decimal
  0   1   1   0   0     =    12
  1   0   1   0   1     =    21
  ----------------------     exclusive or
  1   1   0   0   1     =    25
```

The following unary operator pops the top value from the stack, performs an operation, and pushes the result on the stack:

%~ Bitwise complement. For example, %{25}%~ results in a value of -26, as shown in the following display.

```
                Binary                      Decimal

0   0   0   1   1   0   0   1       =    25

-----------------------------------     Complement

1   1   1   0   0   1   1   0       =    -26
```

Logical Operators

The following logical operators pop the top two values from the stack, perform an operation, and push the logical result (either 0 for false or 1 for true) on the stack:

%= Equal to. For example, if the parameter p1 has the value 3, the expression %p1%{2}%= is equivalent to 3=2 and produces a result of 0 (false).

%> Greater than. For example, if the parameter p1 has the value 3, the expression %p1%{0}%> is equivalent to 3>0 and produces a result of 1 (true).

%< Less than. For example, if the parameter p1 has the value 3, the expression %p1%{4}%< is equivalent to 3<4 and produces a result of 1 (true).

The following unary operator pops the top value from the stack, performs an operation, and pushes the logical result (either 0 or 1) on the stack.

%! Logical negation. This operator produces a value of zero for all non-zero numbers and a value of 1 for zero. For example, %{2}%! results in a value of 0, and %{0}%! results in a value of 1.

Conditional Statements

The condition statement IF-THEN-ELSE has the following format:

%? *expr* %t *thenpart* %e *elsepart* %;

The %e *elsepart* is optional. You can nest conditional statements in the *thenpart* or the *elsepart*.

When INFORMIX-SQL evaluates a conditional statement, it pops the top value from the stack and evaluates it as either `true` or `false`. If the value is `true`, INFORMIX-SQL performs the operations after the %t; otherwise it performs the operations after the %e (if any).

For example, the expression:

```
%?%p1%{3}%=%t;31%;
```

is equivalent to:

```
if p1 = 3 then print ";31"
```

Assuming that p1 has the value `3`, INFORMIX-SQL performs the following steps:

- %? does not perform an operation but is included to make the conditional statement easier to read.
- %p1 pushes the value of p1 on the stack.
- %{3} pushes the value `3` on the stack.
- %= pops the value of p1 and the value `3` from the stack, evaluates the Boolean expression p1=3, and pushes the resulting value `1 (true)` on the stack.
- %t pops the value from the stack, evaluates `1` as `true`, and executes the operations after %t. (Since ";31" is not a stack machine operation, INFORMIX-SQL prints ";31" to the terminal.)
- %; terminates the conditional statement.

Summary of Operators

Figure D-7 summarizes the allowed operations.

Figure D-7
Stack Operations

Operation	Description
%d	write pop() in decimal format
%2d	write pop() in 2-place decimal format
%3d	write pop() in 3-place decimal format
%c	write pop() as a single character
%p[1-9]	push i^{th} parameter
%P[a-z]	pop and store variable
%g[a-z]	get variable and push on stack
%'c'	push char constant
%{n}	push integer constant
%S[a-z]	pop and store static variable
%G[a-z]	get static variable and push
%+	addition. push(pop() op pop())
%-	subtraction. push(pop() op pop())
%*	multiplication. push(pop() op pop())
%/	integer division. push(pop() op pop())
%m	modulus. push(pop() op pop())
%&	bit and. push(pop() op pop())
%|	bit or. push(pop() op pop())
%^	bit exclusive or. push(pop() op pop())
%~	bitwise complement. push(op pop())

(1 of 2)

Operation	Description
%=	equal to. push(pop() op pop())
%>	greater than. push(pop() op pop())
%<	less than. push(pop() op pop())
%!	logical negation. push(op pop())
%? *expr* %t *then part* %e *elsepart* % ;	if-then-else; the %e *elsepart* is optional. else-if's are possible (c's are conditions): %? c1 %t...%e c2 %t...%e c3 %t...%e...%; nested if's allowed.
All other characters are written to the terminal; use '%%' to write '%'.	

(2 of 2)

Examples

To illustrate, consider the monochrome Wyse terminal. Figure D-8 shows the sequences for various display characteristics.

```
ESCAPE G 0   Normal
ESCAPE G 1   Blank(invisible)
ESCAPE G 2   Blink

ESCAPE G 4   Reverse
ESCAPE G 5   Reverse and blank
ESCAPE G 6   Reverse and blink

ESCAPE G 8   Underscore
ESCAPE G 9   Underscore and blank
ESCAPE G :   Underscore and blink

ESCAPE G <   Underscore and reverse
ESCAPE G =   Underscore, reverse, and blank
ESCAPE G >   Underscore, reverse, and blink
```

Figure D-8 *Wyse Escape Sequences*

The characters after G form an ASCII sequence from the character 0 (zero) through ?. You can generate the character by starting with 0 and adding 1 for blank, 2 for blink, 4 for reverse, and 8 for underline.

You can construct the **termcap** entry in stages, as outlined in the following display. %pi refers to pushing the i[th] parameter on the stack. The designation for is \E. The **termcap** entry for the Wyse terminal must contain the following **ZA** entry in order for INFORMIX-SQL monochrome attributes such as REVERSE and BOLD to work correctly:

```
ZA =
\EG#print \EG
%'O'#push 'O' (normal) on the stack
%?%p1%{7}%=%t%{1}%|#if p1 = 7 (invisible), set
#the 1 bit (blank);
%e%p1%{3}%>       #if p1 > 3 and < 7, set the 64 flag (dim);
  %p1%{7}%<%&%t%{64}%|#
     %;%;#
%?%p2%t%{4}%|%;#if p2 is set, set the 4 bit (reverse)
%?%p3%t%{2}%|%;#if p3 is set, set the 2 bit (blink)
%?%p4%t%{8}%|%;#if p4 is set, set the 8 bit (underline)
%c: #print whatever character
    #  is on top of the stack
```

You then concatenate these lines as a single string that ends with a colon and has no embedded NEWLINEs. The actual **ZA** entry for the Wyse 50 terminal follows:

```
ZA = \EG%'O'%?%p1%{7}%=%t%{1}%|%e%p1%{3}%>%p1%{7}%<%&%t%{64}
%|%;%;%?%p2%t%{4}%|%;%?%p3%t%{2}%|%;%?%p4%t%{8}%|%;%c:
```

The next example is for the ID Systems Corporation ID231, a color terminal. On this terminal, to set color and other characteristics you must enclose a character sequence between a lead-in sequence (ESCAPE [0) and a terminating character (m). The first in the sequence is a two-digit number that determines whether the assigned color is in the background (30) or in the foreground (40). The next is another two-digit number that is the other of 30 or 40, incremented by the color number. These characters are followed by 5 if there is blinking and by 4 for underlining. The code in Figure D-9 sets up the entire escape sequence.

```
ZA =
\E[0;#print lead-in
%?%p1%{0}%=%t%{7}#encode color number (translate
%e%p1%{1}%=%t%{3}#   from Figure D-6 to the number
%e%p1%{2}%=%t%{5}#   for the ID231)
%e%p1%{3}%=%t%{1}#
%e%p1%{4}%=%t%{6}#
%e%p1%{5}%=%t%{2}#
%e%p1%{6}%=%t%{4}#
%e%p1%{7}%=%t%{0}%;#
%?%p2%t30;%{40}%+%2d#if p2 is set, print '30' and
    #  '40' + color number (reverse)
%e40;%{30}%+%2d%;#  else print '40' and
    #  '30' + color number (normal)
%?%p3%t;5%;#if p3 is set, print 5 (blink)
%?%p4%t;4%;#if p4 is set, print 4 (underline)
m   #print 'm' to end character
    #  sequence
```

Figure D-9
Sample ZA String for ID231

When you concatenate these strings, the **termcap** entry is as shown in Figure D-10.

```
ZA =\E[0;%?%p1%{0}%=%t%{7}%e%p1%{1}%=%t%{3}%e%p1%{2}%=
%t%{5}%e%p1%{3}%=%t%{1}%e%p1%{4}%=%t%{6}%e%p1%{5}%=%t%
{2}%e%p1%{6}%=%t%{4}%e%p1%{7}%=%t%{0}%;%?%p2%t30;%{40}
%+%2d%e40;%{30}%+%2d%;%?%p3%t;5%;%?%p4%t;4%;m
```

Figure D-10
Concatenated ZA String for ID231

In addition to the **ZA** capability, you can use other **termcap** capabilities. **ZG** is the number of character positions on the screen occupied by the attributes of **ZA**. Like the **sg** numeric capability, **ZG** is not required if no extra character positions are needed for display attributes. The value for the **ZG** entry is usually the same value as for the **sg** entry.

terminfo

If you have set the INFORMIXTERM environment variable to **terminfo**, INFORMIX-SQL uses the **terminfo** directory indicated by the **TERMINFO** environment variable (or **/usr/lib/terminfo** if **TERMINFO** is not set). INFORMIX-SQL uses the information in **terminfo** to draw borders and display certain intensity attributes.

You might want to modify a file in the **terminfo** directory if you want to specify or change the graphics characters used for borders in screen forms.

Tip: *If you use **terminfo** (instead of **termcap**), you cannot use color or certain intensity attributes with INFORMIX-SQL. To use color attributes with INFORMIX-SQL, you must use **termcap**.*

Some terminals cannot support graphics characters. You should read this appendix and the user guide that comes with your terminal to determine whether or not the changes described in this appendix are applicable to your terminal.

To modify a **terminfo** file, you need to be familiar with the following:

- The format of **terminfo** entries
- The **infocmp** program
- The **tic** program

This information is summarized in this appendix; however, you should refer to your operating system documentation for a complete discussion.

Format of a *terminfo* Entry

terminfo is a directory that contains a file for each terminal name that is defined. Each file contains a compiled **terminfo** entry for that terminal. This section describes the general format of **terminfo** entries. For a complete description of **terminfo**, refer to your operating system documentation.

A **terminfo** entry contains a list of names for the terminal, followed by a list of the terminal's capabilities. The three types of capabilities are:

- Boolean
- Numeric
- String

All **terminfo** entries have the following format:

- ESCAPE is specified as a backslash (\ ) followed by the letter E, and CONTROL is specified as a caret (^). Do not use the ESCAPE or CONTROL keys to indicate escape sequences or control characters in a **terminfo** entry.
- Each capability, including the last entry, is followed by a comma (,).

Figure D-11 shows a basic **terminfo** entry for the Wyse 50 terminal.

```
. Entry for Wyse 50:

w5|wy50|wyse50,
  am, cols#80, lines#24, cuul=^K, clear=^Z,
  home=^^, cufl=^L, cup=\E=%p1%'\s'%+%c%p2%'\s'%+%c,
  bw, ul, bel=^G, cr=\r, cud1=\n, cub1=\b, kpb=\b, kcud1=\n,
  kdub1=\b, nel=\r\n, ind=\n,
  xmc#1, cbt=\EI,
```

Figure D-11
Wyse 50 terminfo Entry

Tip: *Comment lines begin with a period (.).*

Terminal Names

A **terminfo** entry starts with one or more names for the terminal (each separated by a vertical bar (|)). For example, the **terminfo** entry for the Wyse 50 terminal starts with the following line:

```
w5|wy50|wyse50,
```

The **terminfo** entry can be accessed using any one of these names.

Boolean Capabilities

A Boolean capability is a two- to five-character code that indicates whether or not a terminal has a specific feature. If the Boolean capability is present in the **terminfo** entry, the terminal has that particular feature.

Figure D-12 shows some of the Boolean capabilities for the Wyse 50 terminal.

```
    bw,am,

.   bwbackward wrap
.   amautomatic margins
```

Figure D-12
Boolean Capabilities for the Wyse 50

Numeric Capabilities

A numeric capability is a two- to five-character code followed by a pound symbol (#) and a value. Figure D-13 shows the numeric capabilities for the number of columns and the number of lines on a Wyse 50 terminal.

```
    cols#80,lines#24,

.   colsnumber of columns in a line
.   linesnumber of lines on the screen
```

Figure D-13
Numeric Capabilities for the Wyse 50

String Capabilities

A string capability specifies a sequence that can be used to perform a terminal operation. A string capability is a two- to five-character code followed by an equal sign (=) and a string ending at the next delimiter (,).

Most **terminfo** entries include string capabilities for clearing the screen, cursor movement, arrow keys, underscore, function keys, and so on. Figure D-14 shows many of the string capabilities for the Wyse 50 terminal.

```
            el=\ET,clear=\E*,
            cuf1=^L,cuu1=^K,
            smso=\EG4,rmso=\EG0,
            kcuu1=^K,kcud1=^J,kcuf1=^L,kcub1=^H,
            kf0=^A@^M,kf1=^AA^M,kf2=^AB^M,kf3=^AC^M,

.   el=\Etclear to end of line
.   clear=\E*clear the screen
.   cuf1=^Lnon-destructive cursor right
.   cuu1=^Kup one line

.   smso=\EG4start stand-out
.   rmso=\EG0end stand-out

.   kcuu1=^Kup arrow key
.   kcud1=^Jdown arrow key
.   kcuf1=^Lright arrow key
.   kcub1=^Hleft arrow key

.   kf0=^A@^Mfunction key F1
.   kf1=^AA^Mfunction key F2
.   kf2=^AB^Mfunction key F3
.   kf3=^AC^Mfunction key F4
```

Figure D-14 *String Capabilities for the Wyse 50*

Specifying Graphics Characters in Screen Forms

INFORMIX-SQL uses characters defined in a **terminfo** file to draw the borders of boxes and other rectangular shapes that appear in a screen form. If no characters are defined in the **terminfo** file, INFORMIX-SQL uses the hyphen (-) for horizontal lines, the vertical bar (|) for vertical lines, and the plus sign (+) for corners.

Look at the **terminfo** source file (using **infocmp**) to see if the entry for your terminal includes these definitions (look for the **acsc** capability, described later in this section). If the file does not contain border character definitions for your terminal type, or if you want to specify alternative border characters, you or your system administrator can modify the **terminfo** source file. Refer to your operating-system documentation for a description of how to decompile **terminfo** entries using the **infocmp** program.

To specify border characters in the terminfo source file for your terminal

1. Determine the escape sequences for turning graphics mode on and off.

   This information is located in the manual that comes with your terminal. For example, on Wyse 50 terminals, the escape sequence for entering graphics mode is ESCAPE H^B and the escape sequence for leaving graphics mode is ESCAPE H^C.

Tip: *Terminals without a graphics mode do not have this escape sequence. The procedure for specifying alternative border characters on a non-graphics terminal is discussed at the end of this section.*

2. Identify the ASCII equivalents for the six graphics characters that INFORMIX-SQL requires to draw the border.

   The ASCII equivalent of a graphics character is the key you would press in graphics mode to obtain the indicated character.

   Figure D-15 shows the graphics characters and the ASCII equivalents for a Wyse 50 terminal.

   Figure D-15
   Wyse 50 ASCII Equivalents for Border Graphics Characters

Window Border Position	Graphics Character	ASCII Equivalent
upper left corner	⌜	2
lower left corner	⌞	1
upper right corner	⌝	3
lower right corner	⌟	5
horizontal	-	z
vertical	|	6

   Again, this information should be located in the manual that comes with your terminal.

3. Edit the **terminfo** source file for your terminal (you can decompile it using **infocmp** redirected to a file).

Tip: *You might want to make a copy of your* ***terminfo*** *directory before you edit files. You can use the* ***TERMINFO*** *environment variable to point to whichever copy of the* ***terminfo*** *directory you want to access.*

Use the format:

```
terminfo-capability=value
```

to enter values for the following **terminfo** capabilities:

smacs The escape sequence for entering graphics mode. In a terminfo file, ESCAPE is represented as a backslash (\ ) followed by the letter E; CONTROL is represented as a caret (^). For example, the Wyse 50 escape sequence ESCAPE-H CONTROL-B is represented as \EH^B.

rmacs The escape sequence for leaving graphics mode. For example, the Wyse 50 escape sequence ESCAPE-H CONTROL-C is represented as \EH^C.

acsc The concatenated, paired list of ASCII equivalents for the six graphics characters used to draw the border. You can specify the characters in any order, but you must pair the ASCII equivalents for your terminal with the following system default characters.

Figure D-16
System Default Characters for Border Positions

Position	System Default Character
upper left corner	l
lower left corner	m
upper right corner	k
lower right corner	j
horizontal	q
vertical	x

Use the following format to specify the **acsc** value:

```
defnewdefnew . . .
```

where *def* is the default character for a particular border character and *new* is that terminal's equivalent for the same border character.

For example, on the Wyse 50 terminal, given the ASCII equivalents in Figure D-15 and the system default characters in Figure D-16, the **acsc** capability would be set as shown in Figure D-17.

```
acsc=l2m1k3j5qzx6
```

Figure D-17
Wyse 50 acsc Setting

4. Use **tic** to recompile the modified **terminfo** file. See your operating-system documentation for a description of the **tic** program.

The following example shows the full setting for specifying alternative border characters on the Wyse 50:

```
smacs=\EH^B,          . sets smacs to ESC H CTRL B
rmacs=\EH^C,          . sets rmacs to ESC H CTRL C
acsc=l2m1k3j5qzx6,    . sets acsc to the ASCII equivalents
                      . of graphics characters for upper
                      . left (l), lower left (m), upper right (k),
                      . lower right (j), horizontal (q),
                      . and vertical (x)
```

If you prefer, you can enter this information in a linear sequence:

```
smacs=\EH^B,rmacs=\EH^C,acsc=l2m1k3j5qzx6,
```

Terminals Without Graphics Capabilities

For terminals without graphics capabilities, you must enter a blank value for the **smacs** and **rmacs** capabilities. For **acsc**, enter the characters you want INFORMIX-SQL to use for the window border.

The following example shows possible values for **smacs**, **rmacs**, and **acsc** in an entry for a terminal without graphics capabilities. In this example, window borders would be drawn using underscores (_) for horizontal lines, vertical bars (|) for vertical lines, periods (.) for the top corners, and vertical bars (|) for the lower corners.

```
smacs=,rmacs=,acsc=l.m|k.j|q_x|,
```

INFORMIX-SQL uses the graphics characters in the **terminfo** file when you specify a screen border in a PERFORM screen.

Color and Intensity

If you use **terminfo**, you cannot use color or the BOLD or BLINK intensity attributes with the COLOR attribute in PERFORM. If you specify these attributes, they are ignored.

If the **terminfo** entry for your terminal contains the **ul** and **so** attributes, you can use the UNDERLINE and REVERSE intensity attributes, however. You can see if your **terminfo** entry includes these capabilities by using the **infocmp** program. Refer to your operating-system documentation for information about **infocmp**.

If you want to use color and intensity in your INFORMIX-SQL screen forms, you must use **termcap** (by setting the **INFORMIXTERM** environment variable to **termcap**, and by setting the **TERMCAP** environment variable to **$INFORMIXDIR/etc/termcap**). For more information, refer to the "Environment Variables" Appendix and the Preface.

Appendix

E

The ASCII Character Set

In the following table, ^ represents the CONTROL key.

Num	Char	Num	Char	Num	Char
0	^@	43	+	86	V
1	^A	44	,	87	W
2	^B	45	-	88	X
3	^C	46	.	89	Y
4	^D	47	/	90	Z
5	^E	48	0	91	[
6	^F	49	1	92	\
7	^G	50	2	93	]
8	^H	51	3	94	^
9	^I	52	4	95	_
10	^J	53	5	96	‘
11	^K	54	6	97	a
12	^L	55	7	98	b
13	^M	56	8	99	c
14	^N	57	9	100	d
15	^O	58	:	101	e
16	^P	59	;	102	f
17	^Q	60	<	103	g
18	^R	61	=	104	h
19	^S	62	>	105	i
20	^T	63	?	106	j
21	^U	64	@	107	k
22	^V	65	A	108	l

(1 of 2)

Num	Char	Num	Char	Num	Char
23	^W	66	B	109	m
24	^X	67	C	110	n
25	^Y	68	D	111	o
26	^Z	69	E	112	p
27	esc	70	F	113	q
28	^\	71	G	114	r
29	^]	72	H	115	s
30	^^	73	I	116	t
31	^_	74	J	117	u
32		75	K	118	v
33	!	76	L	119	w
34	"	77	M	120	x
35	#	78	N	121	y
36	$	79	O	122	z
37	%	80	P	123	{
38	&	81	Q	124	|
39	’	82	R	125	}
40	(	83	S	126	~
41	)	84	T	127	del
42	*	85	U		

(2 of 2)

Appendix

F

Reserved Words

In this release of INFORMIX-SQL, very few words are reserved. You can use the words that were reserved in previous releases of INFORMIX-SQL as identifiers. For example, you can execute a statement such as the following:

```
CREATE TABLE table (column INTEGER,
    date DATE, char CHAR(20))
```

However, using some of the formerly reserved words can cause ambiguities in your INFORMIX-SQL statements. These ambiguities can cause INFORMIX-SQL to:

- process the statement differently than you intended.
- produce an error.

This section discusses these potential ambiguities and syntax errors.

If your table is ANSI compliant, some words are still reserved. For a list of the ANSI reserved words, see the section entitled "Avoiding the ANSI Reserved Words" on page F-9.

Potential Ambiguities and Syntax Errors

Although you can now use the formerly reserved words as identifiers in INFORMIX-SQL statements, syntactic ambiguities can occur. Thus, a statement might not produce the desired results. This section describes:

- using functions as column names.
- using keywords as column names.
- using keywords as table names.
- workarounds that use the keyword AS.

Using Functions as Column Names

When using built-in function names as column names, the database server can interpret the column name as a function. For example, the following statement specifies a column named **avg**. This statement fails because the database server interprets **avg** as an aggregate function rather than as a column name.

```
SELECT avg FROM mytab
```

You can avoid this ambiguity by including a table name with the column name, as shown in the following example:

```
SELECT mytab.avg FROM mytab
```

This ambiguity applies to the aggregate functions (**AVG, COUNT, MAX, MIN, SUM**), the **LENGTH** function, the date functions (**DATE, DAY, MDY, MONTH, WEEKDAY, YEAR**), and the datetime function **EXTEND**. For general descriptions of these functions, refer to the *Informix Guide to SQL: Tutorial*.

If you use the keyword **TODAY, CURRENT**, or **USER** as a column name, ambiguity can also occur, as shown in the following example:

```
CREATE TABLE mytab (user CHAR(10),
    current DATETIME HOUR TO SECOND,today DATE)

INSERT INTO mytab VALUES("josh","11:30:30","1/22/89")

SELECT user,current,today FROM mytab
```

The database server interprets **user**, **current**, and **today** in the SELECT statement as the functions **USER**, **CURRENT**, and **TODAY**. Thus, instead of returning `josh, 11:30:30,1/22/89`, the **SELECT** statement returns the current user name, the current time, and the current date.

If you want to select the actual columns of the table, you must write the SELECT statement in one of two ways:

```
SELECT mytab.user, mytab.current, mytab.today FROM mytab
```

or, equivalently:

```
SELECT * FROM mytab
```

For general descriptions of the **TODAY**, **CURRENT**, and **USER** functions, see the *Informix Guide to SQL: Tutorial*.

Using Keywords as Column Names

INFORMIX-SQL supports specific workarounds for using a formerly reserved keyword as a column name in an INFORMIX-SQL statement. In some cases, INFORMIX-SQL offers more than one workaround. This section describes:

- using ALL as a column name.
- using UNIQUE or DISTINCT as a column name.
- using INTERVAL or DATETIME as a column name.
- using **rowid** as a column name.

Using ALL as a Column Name

Using **all** as a column name causes the following SELECT statement to fail because the database server interprets **all** as a keyword rather than as a column name:

```
SELECT all FROM mytab
```

To include a column name **all** in a SELECT statement, you can include the ALL keyword prior to the **all** column name, as shown in the following example:

```
SELECT ALL all FROM mytab
```

You can also prefix the column name with the table name. For example, you could specify the following:

```
SELECT mytab.all FROM mytab
```

Using UNIQUE or DISTINCT as a Column Name

Using **unique** or **distinct** as a column name causes the CREATE TABLE statement to fail because the database server interprets **unique** as a keyword rather than as a column name:

```
CREATE TABLE mytab (unique INTEGER)
```

You can, however, name a column **unique** by using two statements. The first statement creates the column **mycol**; the second statement renames the column **mycol** to **unique**, as follows:

```
CREATE TABLE mytab (mycol INTEGER)

RENAME COLUMN mytab.mycol TO unique
```

Using INTERVAL or DATETIME as a Column Name

Using **interval** as a column name causes the following SELECT statement to fail because the database server interprets **interval** as a keyword and expects it to be followed by an INTERVAL qualifier:

```
SELECT interval FROM mytab
```

To include a column named **interval** in a SELECT statement, you should preface the column name with the table name, as shown in the following example:

```
SELECT mytab.interval FROM mytab
```

You can also include the owner name as well as the table name:

```
SELECT josh.mytab.interval FROM josh.mytab
```

Using rowid as a Column Name

Every Informix database table has a virtual column named **rowid**. This column contains the record number associated with each row in a table. To avoid ambiguity, you cannot use **rowid** as a column name. The following actions cause an error:

- Creating a table or view with a column named **rowid**.
- Altering a table by adding a column named **rowid**.
- Renaming a column to **rowid**.

You can, however, use the term **rowid** as a table name:

```
CREATE TABLE rowid (column INTEGER,
    date DATE, char CHAR(20))
```

Using Keywords as Table Names

If you use a previously reserved word as the name for a table, some SQL statements might be ambiguous. You can avoid ambiguous statements by prefixing the table name with the name of the table's owner.

For example, using **statistics** as a table name causes the following UPDATE statement to fail because the database server interprets it as part of the UPDATE STATISTICS syntax rather than as a table name in an UPDATE statement:

```
UPDATE statistics SET mycol = 10
```

You can, however, include a table named **statistics** in an UPDATE statement by specifying an owner name with the table name, as shown in the following example:

```
UPDATE josh.statistics SET mycol = 10
```

Using **outer** as a table name causes the following SELECT statement to fail because the database server interprets **outer** as a keyword for performing an outer join:

```
SELECT mycol FROM outer
```

Again, by specifying an owner name with the table name, you can avoid any ambiguity and create a SELECT statement that executes properly, as shown in the following example:

```
SELECT mycol FROM josh.outer
```

Workarounds that Use the Keyword AS

Some formerly reserved words cannot be used as column labels or table aliases. As a workaround, INFORMIX-SQL provides the AS keyword that lets you use these words as column labels and table aliases.

Because the AS keyword syntax is a part of the proposed ANSI SQL2 standard, but is not included in the ANSI SQL89 standard, the database server generates ANSI warnings if you use the AS keyword and one of the following statements is true:

- The **DBANSIWARN** environment variable is set.
- You specify the **-ansi** flag when invoking INFORMIX-SQL.

The syntax for using AS with a column label is as follows:

```
column-name AS display-label FROM table-name
```

The syntax for using AS with a table alias is as follows:

```
SELECT select-list FROM table-name AS table-alias
```

Both of these options are described below.

Using AS With Column Labels

To use one of the following keywords as a column label, you must use the AS keyword:

- ★ AS
- FROM
- UNITS
- YEAR
- MONTH
- DAY
- HOUR

- MINUTE
- SECOND
- FRACTION

For example, the following statement fails because the database server interprets **units** as a DATETIME qualifier for the column named **mycol**:

```
SELECT mycol units FROM mytab
```

By using the keyword AS, however, you can avoid any ambiguity, as shown in the following example:

```
SELECT mycol AS units FROM mytab
```

You must also use the AS keyword to select a column labeled **as** or **from**. For example, the following statement fails because INFORMIX-SQL does not find the required FROM clause. INFORMIX-SQL interprets the *column label* **as** as the *keyword* AS. INFORMIX-SQL then interprets the keyword FROM as the column label to assign to **mycol**:

```
SELECT mycol as FROM mytab
```

By using the keyword AS, however, you can avoid any ambiguity, as shown in the following example:

```
SELECT mycol AS as FROM mytab
```

The following statement fails because the database server expects a table name to follow the first **from**:

```
SELECT mycol from FROM mytab
```

By using the keyword AS, however, you can identify the first **from** as a column label, as shown in the following example:

```
SELECT mycol AS from FROM mytab
```

Using AS with Table Aliases

To use one of the following keywords as a table alias, you must use the AS keyword:

- ORDER
- FOR
- GROUP
- HAVING
- INTO
- UNION
- WHERE

For example, the following statement fails because the database server interprets **order** as part of an ORDER BY clause:

```
SELECT * FROM mytab order
```

By using the keyword AS, however, you can identify **order** as a table alias:

```
SELECT * FROM mytab AS order
```

You must also use the keyword AS to give a table the alias of WITH, CREATE, or GRANT. For example, the following statement fails because the database server interprets **with** as part of the WITH CHECK OPTION syntax:

```
SELECT * FROM mytab with
```

By using the keyword AS, however, you can identify **with** as a table alias, as shown in the following example:

```
SELECT * FROM mytab AS with
```

The following statement fails because the database server interprets the keyword **create** as part of the syntax to create an entity such as a table, synonym, or view:

```
SELECT * FROM mytab create
```

By using the keyword AS, however, you can identify **create** as a table alias, as shown in the following example:

```
SELECT * FROM mytab AS create
```

Avoiding the ANSI Reserved Words

If you have an ANSI-compliant table, several words are reserved. INFORMIX-SQL generates a warning if you use an ANSI reserved word as an identifier in a statement and one or both of the following statements is true:

- The **DBANSIWARN** environment variable is set.
- You specify the **-ansi** flag when invoking INFORMIX-SQL.

The ANSI reserved words are as follows.

all	cursor	goto	of	smallint
and	dec	grant	on	some
any	decimal	group	open	sql
as	declare	having	option	sqlcode
asc	delete	in	or	sqlerror
avg	desc	indicator	order	sum
begin	distinct	insert	pascal	table
between	double	int	pli	to
by	end	integer	precision	union
char	escape	into	privileges	unique
character	exec	is	procedure	update
check	exists	language	public	user
close	fetch	like	real	values
cobol	float	max	rollback	view
commit	for	min	schema	whenever
continue	fortran	module	section	where
count	found	not	select	with
create	from	null	set	work
current	go	numeric		

Appendix

G

Accessing Programs from the Operating System

You can access the modules that comprise INFORMIX-SQL in the following three ways:

- From the INFORMIX-SQL Main menu
- Directly from the operating-system command line using the module names
- Directly from the operating-system command line using a shortened version of the INFORMIX-SQL Main menu options

If you are accessing a program through either the Main menu or the **User-menu** option, and you receive an error message that indicates that you have run out of space in memory, exit to the command line. This action clears memory. Then enter `isql` from the command line and you can resume your operations.

The command-line syntax for accessing each INFORMIX-SQL module is described later in this appendix. The procedure for using the shortened version of the INFORMIX-SQL Main menu from the command line is described in "Accessing FORM Menu Options" on page G-2.

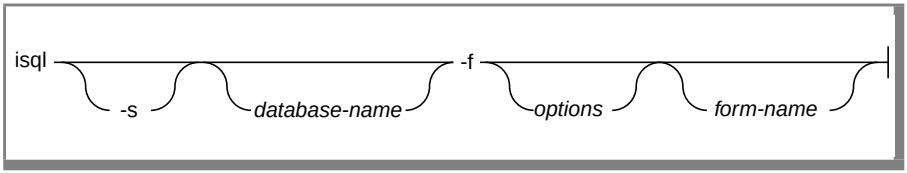
For easy reference, the INFORMIX-SQL Main menu options are shown in Figure G-1 on page G-2.

Figure G-1
INFORMIX-SQL Main menu Options

Form	Run, Modify, Generate, New, Compile, Drop
Report	Run, Modify, Generate, New, Compile, Drop
Query-language	New, Run, Modify, User-editor, Output, Choose, Save, Info, Drop
Database	Create, Drop
User-menu	Run, Modify
Table	Create, Alter, Info, Drop

Accessing FORM Menu Options

The operating system command-line syntax for accessing INFORMIX-SQL FORM menu options is as follows.



isql	is the program call for INFORMIX-SQL.
-s	calls the *silent* option and suppresses all non-essential screen messages.
database-name	is the name of a database in your current directory or a directory cited in your **DBPATH** environment variable.
-f	calls the **Form** option from the INFORMIX-SQL Main menu.
options	are the first letters of the FORM menu options you select. Do not include a blank space between **-f** and any option letters.
form-name	is the name of the form you want to access. Do not include an extension.

INFORMIX-SQL returns you to the operating system after you complete the specified operation.

The following command runs the **customer** form:

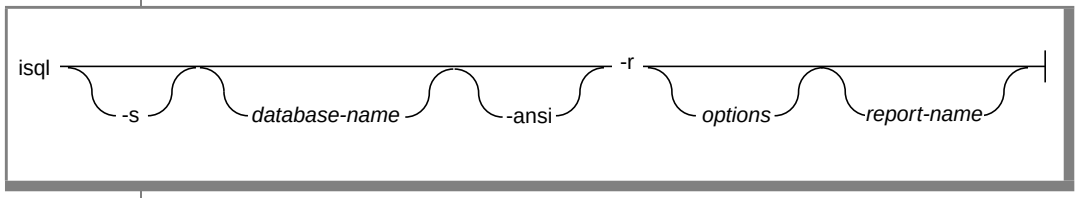
```
isql -fr customer
```

The following generates a form based on the **stores7** demonstration database:

```
isql stores7 -fg
```

Accessing REPORT Menu Options

The operating system command-line syntax for accessing INFORMIX-SQL REPORT menu options is as follows.



isql	is the program call for INFORMIX-SQL.
-s	calls the *silent* option and suppresses all non-essential screen messages.
database-name	is the name of a database in your current directory or a directory cited in your **DBPATH** environment variable.
-ansi	causes INFORMIX-SQL to generate a warning if it encounters an Informix extension to the SELECT statement when compiling your report.
-r	calls the **Report** option from the INFORMIX-SQL Main menu.
options	are the first letters of the REPORT menu options you select. Do not include a blank space between **-r** and any option letters.
report-name	is the name of the report you want to access. Do not include an extension in the report name.

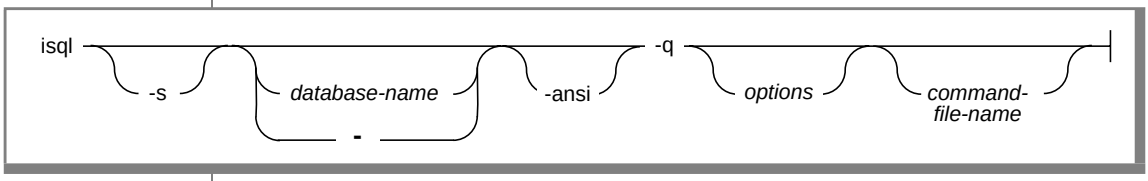
INFORMIX-SQL returns you to the operating system after you complete the specified operation.

The following command compiles the **clist1** report:

```
isql -rc clist1
```

Accessing QUERY-LANGUAGE Menu Options

The operating system command-line syntax for accessing INFORMIX-SQL QUERY-LANGUAGE menu options is as follows.



isql	is the program call for INFORMIX-SQL.
-s	calls the *silent* option and suppresses all non-essential screen messages.
-	indicates that the database name is created or established in the command file.
database-name	is the name of a database in your current directory or a directory cited in your **DBPATH** environment variable.
-ansi	causes INFORMIX-SQL to generate a warning whenever it encounters an Informix extension to ANSI syntax.
-q	calls the **Query-language** option from the INFORMIX-SQL Main menu.
options	are the first letters of the QUERY-LANGUAGE menu options you select. Do not include a blank space between **-q** and any option letters.
command-file	is the name of the **.sql** file that you want to access. Do not include an extension in the command filename.

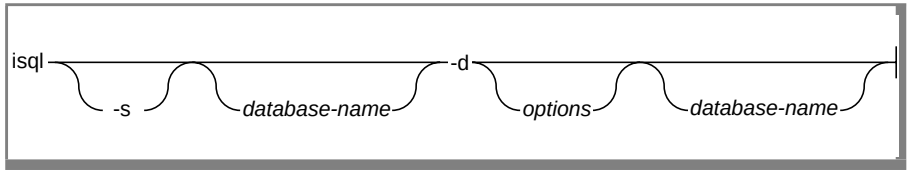
INFORMIX-SQL returns you to the operating system after you complete the specified operation.

The following command chooses the **ex1** file and makes the SQL statements it contains the current statements:

```
isql -s stores7 -qc ex1
```

Accessing DATABASE Menu Options

The operating system command-line syntax for accessing INFORMIX-SQL DATABASE menu options is as follows.



isql	is the program call for INFORMIX-SQL.
-s	calls the *silent* option and suppresses all non-essential screen messages.
database-name	is the name of a database in your current directory or a directory cited in your **DBPATH** environment variable.
-d	calls the **Database** option from the INFORMIX-SQL Main menu.
options	are the first letters of the DATABASE menu options you select. Do not include a blank space between **-d** and any option letters.
database-name	is the name of the database you want to access.

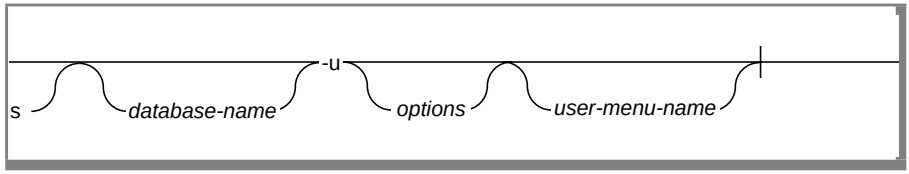
INFORMIX-SQL returns you to the operating system after you complete the specified operation.

The following statement calls the **Select** option on the DATABASE menu:

```
isql -s -ds
```

Accessing USER-MENU Menu Options

The operating system command-line syntax for accessing INFORMIX-SQL USER-MENU menu options is as follows.



isql	is the program call for INFORMIX-SQL.
-s	calls the *silent* option and suppresses all non-essential screen messages.
database-name	is the name of a database in your current directory or a directory cited in your **DBPATH** environment variable.
-u	calls the **User-menu** option from the INFORMIX-SQL Main menu.
options	are the first letters of the USER-MENU menu options you select. Do not include a blank space between **-u** and any option letters.
user-menu-name	is the name of the User-menu you want to run.

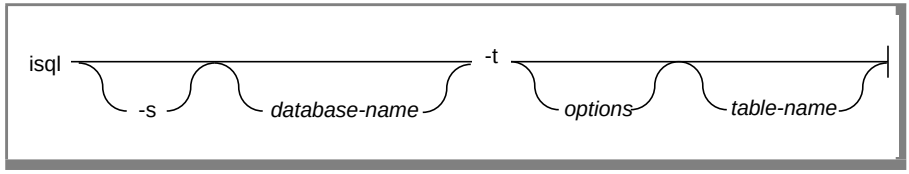
INFORMIX-SQL returns you to the operating system after you complete the specified operation.

The following command runs the User-menu for the **stores7** demonstration database:

```
isql stores7 -ur
```

Accessing TABLE Menu Options

The operating system command-line syntax for accessing INFORMIX-SQL TABLE menu options is as follows.



isql	is the program call for INFORMIX-SQL.
-s	calls the *silent* option and suppresses all nonessential screen messages.
database-name	is the name of a database in your current directory or a directory cited in your **DBPATH** environment variable.
-t	calls the **Table** option from the INFORMIX-SQL Main menu.
options	are the first letters of the TABLE menu options you select. Do not include a blank space between **-t** and any option letters.
table-name	is the name of the table you want to access.

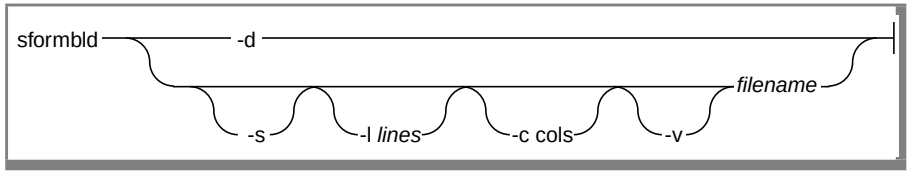
INFORMIX-SQL returns you to the operating system after you complete the specified operation.

The following command creates a table in the **stores7** database:

```
isql -s stores7 -tc
```

FORMBUILD

The command syntax for compiling a customized screen form directly from the operating system is as follows.



sformbld	is the program call for FORMBUILD.
-s	calls the *silent* option and suppresses all non-essential screen messages.
-l *lines*	are optional symbols and an integer to specify the total number of lines of characters (measured vertically) that the terminal can display. (The default is 24.)
-c *cols*	are optional symbols and an integer to specify the width of the screen, in characters. (The default is the number of characters in the longest line of the screen layout, as specified in the SCREEN section.)
-v	tells FORMBUILD to verify that the fields contained in the screen section of the form specification are consistent with the field widths of the corresponding columns.

FORMBUILD reports any discrepancies in the file *filename***.err**.

filename	is the name of the form specification file. Do not include the **.per** extension (*filename***.per**) on the command line.
-d	replaces *filename* and instructs FORMBUILD to prompt you for the information required to create and compile a default form specification.

The **-d** option causes FORMBUILD to construct a `SCREEN SIZE 20` statement to emphasize the default size.

Usage

If the compilation is successful, FORMBUILD creates a compiled form specification named *filename***.frm**. You can use this compiled form specification with PERFORM as a screen form. If the compilation is unsuccessful, FORMBUILD creates an error file named *filename***.err**. You must edit the error file, remove the error messages, and recompile with FORMBUILD before you can use the screen form.

You can also create a customized screen form directly from the operating system using the shortened version of the INFORMIX-SQL Main menu options. This method is described earlier in this appendix.

PERFORM

The command syntax for running a compiled screen form directly from the operating system is as follows.

sperform	is the program call for PERFORM.
filename	is the name of the compiled form specification file. Do not include the **.frm** extension (*filename***.frm**) on the command line.

Usage

The maximum number of *filenames* you can include on the command line is operating-system dependent.

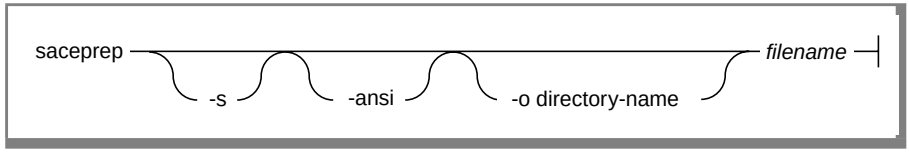
PERFORM displays each form in the order that it appears on the command line.

If PERFORM cannot display a form, it aborts. When multiple filenames are included on the command line, subsequent filenames are not displayed.

You can also run a compiled screen form directly from the operating system using the shortened version of the INFORMIX-SQL Main menu options. This method is described earlier in this appendix.

ACEPREP

The command syntax for compiling a customized report form directly from the operating system is as follows.



saceprep	is the program call for ACEPREP.
-s	calls the *silent* option and suppresses all non-essential screen messages.
-ansi	tells ACEPREP to generate a warning when it encounters an Informix extension to the SELECT statement in your report.
-o *directory-name*	tells ACEPREP to place the output file (either the compiled report specification or the error file) in the indicated directory.
filename	is the name of the report specification file. Do not include the **.ace** extension (***filename*.ace**) on the command line.

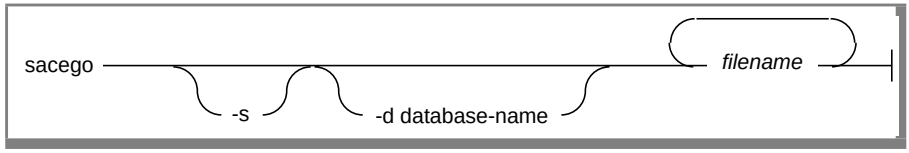
Usage

If the compilation is successful, ACEPREP creates a compiled report specification file named ***filename*.arc**. You can use this compiled report specification with ACEGO to produce a report. If the compilation is unsuccessful, ACEPREP creates an error file named ***filename*.err**. You must edit the error file, remove the error messages, and recompile with ACEPREP before you can run the report.

You can also compile a customized report form directly from the operating system using the shortened version of the INFORMIX-SQL Main menu options. This method is described in "Accessing REPORT Menu Options" on page G-3.

ACEGO

The command syntax for running a compiled report directly from the operating system is as follows.



sacego	is the program call for ACEGO.
-s	calls the *silent* option and suppresses all non-essential screen messages.
-d *database-name*	overrides the database that is named in the report specification and substitutes *database-name*.
filename	is the name of the compiled report specification. Do not include the **.ace** extension (*filename***.ace**) on the command line.

Usage

The maximum number of *filenames* you can include on the command line is operating-system dependent.

ACEGO executes each report in the order in which it appears on the command line.

If ACEGO cannot execute a report, it aborts. When multiple filenames are included on the command line, subsequent filenames are not executed.

You can also run a compiled report directly from the operating system using the shortened version of the INFORMIX-SQL Main menu options. This method is described in "Accessing REPORT Menu Options" on page G-3.

Appendix

H

Notices

IBM may not offer the products, services, or features discussed in this document in all 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
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

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

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and / or changes in the product(s) and / or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J74 / G4
555 Bailey Ave
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.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

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.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

> © (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. (enter the year or years). All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

AIX; DB2; DB2 Universal Database; Distributed Relational Database Architecture; NUMA-Q; OS/2, OS/390, and OS/400; IBM Informix®; C-ISAM®; Foundation.2000™; IBM Informix® 4GL; IBM Informix® DataBlade® Module; Client SDK™; Cloudscape™; Cloudsync™; IBM Informix® Connect; IBM Informix® Driver for JDBC; Dynamic Connect™; IBM Informix® Dynamic Scalable Architecture™ (DSA); IBM Informix® Dynamic Server™; IBM Informix® Enterprise Gateway Manager (Enterprise Gateway Manager); IBM Informix® Extended Parallel Server™; i.Financial Services™; J/Foundation™; MaxConnect™; Object Translator™; Red Brick Decision Server™; IBM Informix® SE; IBM Informix® SQL; InformiXML™; RedBack®; SystemBuilder™; U2™; UniData®; UniVerse®; wintegrate® are trademarks or registered trademarks of International Business Machines Corporation.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Windows, Windows NT, and Excel are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

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

Other company, product, and service names used in this publication may be trademarks or service marks of others.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z