INFORMIX–4GL

SQL-BASED
APPLICATION DEVELOPMENT LANGUAGE

REFERENCE MANUAL
Volume One

Copyright © 1987
Informix Software, Inc.
June 1987

Part No. 200-501-0002-0
Rev. A
PRINTED IN U.S.A.

INFORMIX-4GL
Version 1.10

THE INFORMIX SOFTWARE AND USER MANUAL ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE INFORMIX SOFTWARE AND USER MANUAL IS WITH YOU. SHOULD THE INFORMIX SOFTWARE AND USER MANUAL PROVE DEFECTIVE, YOU (AND NOT INFORMIX OR ANY AUTHORIZED REPRESENTATIVE OF INFORMIX) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR, OR CORRECTION. IN NO EVENT WILL INFORMIX BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS, OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE SUCH INFORMIX SOFTWARE OR USER MANUAL, EVEN IF INFORMIX OR AN AUTHORIZED REPRESENTATIVE OF INFORMIX HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY. IN ADDITION, INFORMIX SHALL NOT BE LIABLE FOR ANY CLAIM ARISING OUT OF THE USE OF OR INABILITY TO USE SUCH INFORMIX SOFTWARE OR USER MANUAL BASED UPON STRICT LIABILITY OR INFORMIX'S NEGLIGENCE. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS, WHICH VARY FROM STATE TO STATE.

All rights reserved. No part of this work covered by the copyright hereon may be reproduced or used in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems—without permission of the publisher.

Published by:
Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025

INFORMIX is a registered trademark of Informix Software, Inc. **RDSQL**, **C-ISAM**, and **File-it!** are trademarks of Informix Software, Inc.

UNIX is a trademark of AT&T.
MS is a trademark of Microsoft Corporation.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subdivision (b)(3)(ii) of the Rights in Technical Data and Computer Software Clause at 52.227-7013 (and any other applicable license provisions set forth in the Government contract).

Copyright © 1981-1987 by Informix Software, Inc.

NOTE: ANY REFERENCES IN DOCUMENTATION TO "RELATIONAL DATABASE, INC." OR "RDS" SHALL MEAN INFORMIX SOFTWARE, INC.

Table of Contents

VOLUME ONE

Introduction

About This Manual 5
Chapter Summary 6
Syntax Conventions 8
Getting Started with INFORMIX-4GL on DOS Systems 10
The Demonstration Database 12

Chapter 1. INFORMIX-4GL Programming

Chapter Overview 1-5
Language Conventions 1-5
- Comments 1-5
- INFORMIX-4GL Identifiers 1-6
- Scope of Identifiers 1-6
- Constants 1-7

Program Variables 1-9
- Data Types 1-9
- Data Conversion 1-12
- Operators and Expressions 1-14
- Binding to Database and Forms 1-18
- The THRU Keyword and the .* Notation 1-19

INFORMIX-4GL Statements 1-21
- Variable Definition 1-21
- Assignment 1-22
- Program Organization 1-22
- Program Flow 1-24
- Screen Interaction 1-25
- Report Generation 1-27

Error and Exception Handling 1-28
Data Validation 1-31
Built-in Functions 1-32
C Functions 1-57
Compiling and Executing Programs 1-63
Program Filename Extensions 1-65

Chapter 2. Using RDSQL

Chapter Overview 2-5
Relational Databases 2-6
RDSQL Identifiers 2-6
Database Data Types 2-7
RDSQL Statement Summary 2-10
Data Definition 2-10
Data Manipulation 2-13
Cursor Management 2-14
SELECT Cursors 2-14
INSERT Cursors 2-26
Dynamic Management 2-30
Preparing Statements 2-31
Executing Prepared Statements 2-35
Data Access 2-42
Data Integrity 2-43
Transactions 2-43
Transaction Log File Maintenance 2-46
Audit Trails 2-47
Comparison of Transactions and Audit Trails 2-48
User Status and Privileges 2-49
Indexing Strategy 2-51
Auto-Indexing 2-53
Clustered Indexes 2-53
Locking 2-54
Row-Level Locking 2-55
Table-Level Locking 2-56
Wait for Locked Row 2-57
NULL Values 2-58
Default Values 2-58
The NULL in Expressions 2-59
The NULL in Boolean Expressions 2-60

The NULL in WHERE Clauses 2-61
The NULL in ORDER BY Clauses 2-62
The NULL in GROUP BY Clauses 2-62
The NULL Keyword in INSERT and UPDATE Statements 2-63
Views 2-64
Creating and Deleting Views 2-65
Querying Through Views 2-66
Modifying Through Views 2-66
Privileges with Views 2-68
Data Constraints Using Views 2-68
Outer Joins 2-70
Table Access by Row ID 2-71
SQLCA Record 2-72
TODAY and USER Functions 2-75

Chapter 3. Form Building and Compiling

Chapter Overview 3-5
Structure of a Form Specification File 3-6
Database Section 3-7
Screen Section 3-8
Tables Section 3-12
Attributes Section 3-14
Instructions Section 3-47
Default Screen Attributes 3-53
Creating and Compiling a Form 3-57
Through the Programmer's Environment 3-57
Through the Operating System 3-58
Using PERFORM Forms in INFORMIX-4GL 3-60

Chapter 4. Report Writing

Chapter Overview 4-5
Calling a REPORT Routine 4-6
Structure of a REPORT Routine 4-7
DEFINE Section 4-8
OUTPUT Section 4-9
ORDER BY Section 4-21

FORMAT Section 4-22
Control Blocks 4-28
Statements 4-46
Expressions and Built-in Functions 4-56

Chapter 5. 4GL Function Library

Chapter Overview 5-5

Chapter 6. Programmer's Environment

Chapter Overview 6-5
The INFORMIX-4GL Menu 6-6
The MODULE Design Menu 6-7
The Modify Option 6-7
The New Option 6-10
The Compile Option 6-10
The Program__Compile Option 6-11
The Run Option 6-11
The Exit Option 6-12
The FORM Design Menu 6-12
The Modify Option 6-12
The Generate Option 6-14
The New Option 6-14
The Compile Option 6-15
The Exit Option 6-15
The PROGRAM Design Menu 6-16
The Modify Option 6-16
The New Option 6-19
The Compile Option 6-19
The Planned__Compile Option 6-20
The Run Option 6-21
The Drop Option 6-21
The Exit Option 6-22
The QUERY LANGUAGE Menu 6-22

VOLUME TWO

Chapter 7. INFORMIX-4GL Statement Syntax

Types of Statements 7-7
Definition of Statements 7-10

Appendix A. Demonstration Database and Application

Appendix B. System Catalogs

Appendix C. Environment Variables

Appendix D. Reserved Words

Appendix E. The *upscol* Utility

Appendix F. The *bcheck* Utility

Appendix G. The *mkmessage* Utility

Appendix H. The *sqlconv* Utility

Appendix I. The *dbupdate* Utility

Appendix J. The *dbload* Utility

Appendix K. DECIMAL Functions for C

Appendix L. Complex Outer Joins

Appendix M. ASCII Character Set

Appendix N. Termcap Changes for INFORMIX-4GL

Appendix O. The *dbschema* Utility

INFORMIX-4GL Error Messages

INFORMIX-TURBO and MS-NET Error Messages

Index

Introduction

Introduction Table of Contents

About This Manual 5

Chapter Summary 6

Syntax Conventions 8

Getting Started with INFORMIX-4GL on DOS Systems 10
  Loading the Database Agent 10
  Entering INFORMIX-4GL 11

The Demonstration Database 12

About This Manual

Relational Database Systems, Inc., (RDS) developed INFORMIX-4GL (*Fourth-Generation Application Development Language*) for the database designer who wants to create custom applications of database management. INFORMIX-4GL contains preprocessors, interpreters, and compilers that allow you to

- Embed database creation and query statements (RDSQL) in a fourth-generation language (INFORMIX-4GL).
- Create interactive screen forms that provide an interface between the user of your application and the database.
- Design output reports that can list and summarize database information.

The documentation for INFORMIX-4GL consists of two parts: the *INFORMIX-4GL User Guide* that introduces both RDSQL and INFORMIX-4GL in stages through example programs that increase in sophistication and subtlety, and the *INFORMIX-4GL Reference Manual* that summarizes all the syntax, rules, and definitions of the variables, statements, and keywords.

This manual assumes that you have used INFORMIX-4GL and are familiar with the structure of relational databases. Those who are acquainted with the programs of INFORMIX-SQL or INFORMIX-ESQL/C, other members of the RDS family of products, will recognize old friends in a new setting. You can read about INFORMIX-SQL in the *INFORMIX-SQL User Guide* and INFORMIX-ESQL/C in the *INFORMIX-ESQL/C Programmer's Manual.*

The underlying file and indexing structure of the database tables created through INFORMIX-4GL, INFORMIX-SQL, or INFORMIX-ESQL/C is built on C-ISAM. For more information about this indexed sequential access method, see the *C-ISAM Programmer's Manual.*

Chapter Summary

The *INFORMIX-4GL Reference Manual* is divided into the following sections:

Introduction	contains a description of the notational conventions used in syntax statements. It also includes a brief description of the demonstration database.
Chapter 1	gives the rules for programming in **INFORMIX-4GL**. It defines the data types and binding of program variables, describes expressions and functions, and explains error handling and the interrelationships among all the **INFORMIX-4GL** statements.
Chapter 2	describes how to interact with databases using **RDSQL**, the RDS extension to the standard Structured Query Language (SQL). This chapter also explains the interrelationships among the various **RDSQL** statements.
Chapter 3	describes the procedure for constructing and compiling screen forms.
Chapter 4	describes the procedure for constructing and compiling reports.
Chapter 5	describes the functions in the **INFORMIX-4GL** library.
Chapter 6	describes the **INFORMIX-4GL** programming environment.
Chapter 7	contains an alphabetized description of each of the **RDSQL** and **INFORMIX-4GL** statements that you can use in an **INFORMIX-4GL** program. Use this chapter as a reference for syntax and rules concerning the use of these statements.

Appendix A	describes the demonstration database used in this manual.
Appendix B	describes the system catalogs that form the data dictionary for a database.
Appendix C	describes the environment variables used by **INFORMIX-4GL**.
Appendix D	lists the reserved words of **INFORMIX-4GL**.
Appendix E	describes the **upscol** utility that permits you to establish default values for the attributes of the fields of your screen forms.
Appendix F	describes the **bcheck** utility that checks and restores the integrity of your index files.
Appendix G	describes the **mkmessage** utility that compiles the help messages for your **INFORMIX-4GL** application.
Appendix H	describes the **sqlconv** utility that converts an **INFORMIX** database to an **RDSQL**-compatible database.
Appendix I	describes the **dbupdate** utility that updates an **RDSQL** Version 1 database to an **RDSQL** Version 2 database.
Appendix J	describes the **dbload** utility that allows you to load data from other database systems or from raw data files into **INFORMIX-4GL** databases.
Appendix K	contains descriptions of C functions that handle DECIMAL type variables in C programs.
Appendix L	amplifies the discussion in the manual on outer joins.
Appendix M	lists the ASCII characters in order.

Appendix N	describes modifications you can make on UNIX systems to your **termcap** file to extend function key definitions, to specify characters for window borders, and to enable INFORMIX-4GL programs to interact in color with your terminal.
Appendix O	describes the **dbschema** utility that you can use to output the RDSQL statements necessary to replicate an individual table or an entire database.
Error Messages	contains an extensive listing of error codes, explains their meaning, and suggests remedies for correcting the errors.

Syntax Conventions

This section explains how to interpret the listings of statement syntax that appear throughout this manual.

ABC — Enter any term in the syntax in uppercase letters exactly as shown, disregarding case. Such terms are keywords. For example,

CREATE INDEX *indname*

means you must enter CREATE INDEX or create index without adding or deleting spaces or letters.

abc — Substitute a value for any term that appears in lowercase italic letters. In the previous example, you should substitute a value for *indname*. In each statement description in Chapter 7, the section "Explanation" describes what values you can substitute for italicized terms.

( ) — Enter parentheses as shown. They are part of the syntax of a statement, not special symbols.

[ ]	Do not enter brackets as part of a statement; they surround any part of a statement that is optional. For example, `CREATE [UNIQUE] INDEX` indicates that you may enter either CREATE INDEX or CREATE UNIQUE INDEX.
|	Select one of the options shown. The vertical bar indicates a choice among several options. For example, `[ONE | TWO [THREE] | FOUR]` means that you can enter either ONE or TWO or FOUR and that, if you enter TWO, you may also enter THREE. (Because the choices are enclosed in square brackets, you can also choose to omit them completely.)
{ }	Choose one of the listed options. When the options are enclosed in braces and separated by vertical bars, you must select one of the options. For example, `{ONE | TWO | THREE}` means that you must enter ONE, or TWO, or THREE and that you may not enter more than one selection.
<u>ABC</u>	Omit or use an option that is underlined. When one of several options is the default option, it appears underlined. For example: [CHOCOLATE | <u>VANILLA</u> | STRAWBERRY] means that you may select any of the three options, but that if you do not enter one of them, VANILLA is the default.

...	Enter additional items like those preceding the ellipsis, if you want. The ellipsis indicates that the immediately preceding item may be repeated indefinitely. For example,

```
statement
...
```

means that there can be a series of statements following the one that is listed.

Getting Started with INFORMIX-4GL on DOS Systems

To start working with INFORMIX-4GL on DOS systems, be sure that your computer is up and running, and that the operating system prompt appears on your screen.

To compile or execute programs that use INFORMIX-4GL, you must perform two steps:

1. Load the *database agent*. The database agent is the part of INFORMIX-4GL that performs the data access and file-manipulation tasks necessary to complete your database management operations.
2. Compile and run your INFORMIX-4GL programs.

Loading the Database Agent

To load the database agent, enter

```
startsql
```

You need to execute this command only once per session. The database agent remains in memory until you execute the command

```
exit
```

from the command line. When you execute this command, it removes the database agent from memory. Because the database agent requires space in memory, you should remove it when you have completed all the desired INFORMIX-4GL, or other SQL, operations. This frees memory for running other programs on your system.

If you attempt to load the database agent when it is already installed in memory, INFORMIX-4GL issues the message STARTSQL has already been executed.

Entering INFORMIX-4GL

After you have loaded the database agent, enter

```
i4ql
```

If you try to run i4ql without first loading the database agent, INFORMIX-4GL issues the message Please run STARTSQL.

If you want to check to see if the database agent has been loaded, enter

```
set
```

This gives you a list of your environment variables. If the database agent has been loaded, the variable SQLCADDR is listed. The following is an example of SQLCADDR as it appears in a list of environment variables. The numbers represent the address in memory where the database agent is loaded.

```
SQLCADDR=333744a8
```

The Demonstration Database

Most of the examples in this manual are based on the **stores** demonstration database. This database, described and listed in detail in Appendix A, involves a wholesale sporting goods firm that maintains a stock of equipment and fills orders to retailers. You can create the **stores** database in the current directory by typing `i4gldemo`. This shell script removes any database labeled **stores** and installs the demonstration database.

The **stores** database contains six tables:

customer	contains information about the retail stores that purchase sporting supplies.
orders	contains information about the individual orders from the retail stores.
items	contains information about the items that make up an order.
stock	contains information about the variety of sporting goods available.
manufact	contains information about manufacturers of sporting goods.
state	contains the names and abbreviations of the states.

hapter 1

INFORMIX-4GL Programming

Chapter 1 Table of Contents

Chapter Overview 5

Language Conventions 5
  Comments 5
  INFORMIX-4GL Identifiers 6
  Scope of Identifiers 6
  Constants 7
    String Constants 7
    Integer Constants 7
    Floating Numeric Constants 8
    Date Constants 8

Program Variables 9
  Data Types 9
    SMALLINT 9
    INTEGER 9
    DECIMAL[(*m*[, *n*])] 10
    SMALLFLOAT 10
    FLOAT 10
    MONEY[(*m*[, *n*])] 10
    CHAR(*n*) 11
    DATE 11
    LIKE *table.column* 11
    RECORD 11
    ARRAY[*i*, *j*, *k*] OF *type* 12
  Data Conversion 12
  Operators and Expressions 14
    Numeric Expressions 15
    String Expressions 15
    Boolean Expressions 16
    Expressions in INFORMIX-4GL Statements 17
  Binding to Database and Forms 18
  The THRU Keyword and the .* Notation 19

INFORMIX-4GL Statements 21
  Variable Definition 21
  Assignment 22
  Program Organization 22

Program Flow 24
Screen Interaction 25
Report Generation 27
Error and Exception Handling 28
Error Handling 29
Exception Handling 31
Data Validation 31

Built-in Functions 32
ASCII 33
CLIPPED 35
COLUMN 37
DATE 39
DATE() 40
DAY() 41
MDY() 42
MONTH() 43
SPACES 44
TODAY 45
USING 46
WEEKDAY() 55
YEAR() 56

C Functions 57
Examples 60

Compiling and Executing Programs 63

Program Filename Extensions 65

Chapter Overview

An INFORMIX-4GL program consists of a series of English-like statements that obey a well-defined syntax. This chapter describes the syntax rules that apply and gives an overview of those statements that do not apply directly to a database. It also describes how the program statements are compiled into an executable program.

Language Conventions

The INFORMIX-4GL programming language contains identifiers, keywords, constants, operators, and expressions. It is completely free-form, like C or Pascal, and ignores extra spaces, tabs, new lines, and comments. When necessary, it uses the keyword END in association with the statement type to terminate a statement; otherwise, it has no statement terminators such as the semicolon.

Comments

Comments in an INFORMIX-4GL program begin with the left brace { and end with the right brace }. You may, alternatively, use the # sign to begin a comment. In this case, the comment terminates at the end of the line. You may not nest comments.

INFORMIX-4GL Identifiers

INFORMIX-4GL deals with a number of different kinds of objects. These include local and global program variables, constants, screen forms, functions, and reports. With the exception of constants, all of these must have an **INFORMIX-4GL** identifier as a name. An **INFORMIX-4GL** identifier is a sequence of letters, digits, and underscores (_). The first character must be a letter or an underscore. Only the first eight characters are significant, and the case of letters is ignored. Identifiers must not be the same as keywords (see Appendix D for a complete list of keywords).

INFORMIX-4GL identifiers may be the same as **RDSQL** identifiers, but such use requires special attention when you use the identifier in an **RDSQL** statement (see Chapter 2 for a discussion of **RDSQL** statements, objects, and identifiers).

Scope of Identifiers

Program variables may be either local or global. Local variables must be defined within a routine and have a scope only for that routine. You must define global variables either prior to the main program module or in a separate globals file. All program files using those global variables must include the statement GLOBALS "*globals-filename*", where *globals-filename* contains the definitions of the global variables.

Forms, cursors, functions, reports, and some statements have identifiers but are not program variables (see Chapter 2 for the definition of cursors and statement identifiers). The scope for identifiers of functions and reports is the entire program. The scope for form, cursor, and statement identifiers is the entire file in which they are used.

Constants

There are several kinds of constants: strings, integers, floating numbers, and dates. INFORMIX-4GL provides three pre-defined constants: TRUE = 1, FALSE = 0, and NOTFOUND = 100. It also permits the assignment of the value NULL to variables and to database columns. NULL values are distinct from blank strings or zeros for numeric variables and columns and represent unknown values. See the section "NULL Values" in Chapter 2 for details about the behavior of NULL values in expressions.

String Constants

String constants are sequences of up to 80 characters enclosed within double quotes. The constant must be written on a single line (no embedded new lines). You may create longer string variables by concatenating string constants. To include a double quote in a string, precede the double quote with the backslash or repeat the double quote:

```
"Enter \"y\" to select this row"
```

or

```
"Enter ""y"" to select this row"
```

The single quote has no special significance in string constants.

Integer Constants

Integer constants must be expressed in decimal notation without embedded commas and without a decimal point. You may precede the integer with a minus or plus sign, but there may be no space, tab, or new line between the sign and the first digit:

```
15        -12        13938
```

Floating Numeric Constants

Non-integer numeric constants are expressed only in base 10 with a decimal point. You may use exponential notation as well:

```
123.456 = 1.23456e2 = 123456.0e-3
```

Date Constants

DATE constants must be enclosed within double quotation marks and can be expressed either with the format *mm/dd/yy* or with *mm/dd/yyyy*. *mm* stands for the month (1 or 01 for January, 2 or 02, for February, and so on). *dd* stands for the day of the month (from 1 to the maximum for that particular month). *yy* and *yyyy* stand for the year. When you use only two digits for the year, **INFORMIX-4GL** assumes that you intend to enter a year beginning with the digits "19."

Program Variables

Information transfer among a screen, report, and database occurs through program variables. You must define the data storage required by a program variable before you can use that variable in an INFORMIX-4GL program. You do this by assigning an identifier to the variable and associating it with a data type, using the DEFINE statement (see Chapter 7 for details).

Data Types

INFORMIX-4GL variables must have one of the following data types:

SMALLINT	CHAR[(*n*)]
INTEGER	DATE
DECIMAL[(*m*[, *n*])]	LIKE *table.column*
SMALLFLOAT	RECORD [LIKE *table.**]
FLOAT	ARRAY[*i,j*, *k*] OF *type*
MONEY[(*m*[, *n*])]	

SMALLINT

These are whole numbers in the range from −32,767 to +32,767.

INTEGER

These are whole numbers in the range from −2,147,483,647 to +2,147,483,647.

DECIMAL[(m[, n])]

These are decimal floating-point numbers with a total of m ($<= 32$) significant digits (the precision) and n ($<= m$) digits to the right of the decimal point (the scale). When you give values for both m and n, the decimal variable obeys fixed-point arithmetic. All numbers less than 0.5×10^{-n} in absolute value have the value zero. The largest absolute value of a variable of this type that can be stored without an error is $10^{m-n} - 10^{-n}$.

The second parameter is optional and, if missing, **INFORMIX-4GL** treats the variable as a floating decimal. This means that DECIMAL(m) variables have a precision of m and a range in absolute value from 10^{-128} to 10^{126}. When printed without other formatting instructions, DECIMAL(m) variables have two decimal places to the right of the decimal point (and an exponent, if necessary). If you designate no parameters, **INFORMIX-4GL** treats DECIMAL as DECIMAL(16), a floating decimal.

SMALLFLOAT

These are binary floating-point numbers corresponding to the float C data type.

FLOAT

These are binary floating-point numbers corresponding to the double C data type.

MONEY[(m[, n])]

Like the DECIMAL data type, the MONEY data type can also take two parameters. The limitation on values for columns of type MONEY(m, n) is the same as for columns of type DECIMAL(m, n). The difference between them is that MONEY type variables are displayed with a dollar sign. The type MONEY(*m*) is defined as DECIMAL(*m*, 2) and, if no parameter is given, MONEY is taken to be DECIMAL(16, 2). Regardless of the number of parameters, **INFORMIX-4GL** always treats the data type MONEY as a fixed decimal number.

CHAR(*n*)

These are character strings of length *n* (where $1 <= n <=$ 32,767). You can refer to substrings of CHAR type variables in LET, ERROR, MESSAGE, and PROMPT statements with the notation *char-variable[m,n]* which selects the *m*th through the *n*th components of the variable *char-variable.*

DATE

These are dates entered as a character string in one of the formats described earlier in the section "Date Constants." **INFORMIX-4GL** stores a DATE variable as an integer whose value is the number of days since December 31, 1899.

LIKE *table.column*

You may define the data type of a program variable indirectly by indicating that it should have the same data type as a column in the database. If the column has type SERIAL (see Chapter 2 for the definition of this data type), **INFORMIX-4GL** assigns the variable the data type INTEGER, but enforces none of the other restrictions on SERIAL data types.

RECORD

This data type describes a set of variables of possibly differing data types, including other records. You can refer to individual elements as *record_name.element_name* and, under most circumstances, to the entire set as *record-name.**. See the section "The THRU Keyword and the .* Notation," later in this chapter, for the limitations on using *record-name.**.

You may define a record by listing its elements and their types or by defining it to be LIKE *table.**, where *table* is a table in the database. If defined LIKE *table.**, the elements of the record have the same names as the columns of *table* and the same corresponding data types.

ARRAY[*i*, *j*, *k*] OF *type*

This is a data type that describes $i \times j \times k$ variables of the same data type. ARRAY variables can have from one to three dimensions. You can have an array of records, but not an array of arrays. The square brackets are required and do not represent an option.

If *char-array*[*i,j,k*] is an array of CHAR type, you select a substring of one of its components with the *char-array*[*i,j,k*][*m,n*] notation. In this example, *i,j,k* are indexes into the array, *m* is the starting position of the substring, and *n* is the stopping position of the substring.

Data Conversion

INFORMIX-4GL performs data-type conversion without objection whenever the process makes sense. You may assign a numeric expression to a CHAR variable and INFORMIX-4GL converts the resulting number to a string. If you use a string representation of a number or a date in an arithmetic expression, INFORMIX-4GL converts the string or date to an appropriate number. INFORMIX-4GL produces an error message only if the conversion could not be made. For example, INFORMIX-4GL converts the string "123.456" to the number 123.456 in an addition, while the string "John" produces an error.

INFORMIX-4GL carries out all arithmetic in an arithmetic expression in type DECIMAL. The type of the resulting variable determines the format of the stored or printed result. The following rules apply to the precision and scale of the DECIMAL variable that results from an arithmetic operation on two numbers:

- All operands, if not already DECIMAL, are converted to DECIMAL, and the resulting number is DECIMAL.

Convert Type	**To**
FLOAT	DECIMAL(16)
SMALLFLOAT	DECIMAL(8)
INTEGER	DECIMAL(10,0)
SMALLINT	DECIMAL(5,0)

- The precision and scale of the result of an arithmetic operation depend on the precision and scale of the operands and on the type of arithmetic expression. The rules are summarized in the table at the end of this section for arithmetic operations on operands with definite scale. When one of the operands has no scale (floating decimal), the result is a floating decimal.

- If the operation is addition or subtraction, INFORMIX-4GL adds trailing zeros to the operand with the smallest scale until the scales are equal.

- If the type of the result of an arithmetic operation requires the loss of significant digits, INFORMIX-4GL reports an error.

- Leading or trailing zeros are not considered significant digits and do not contribute to the determination of precision and scale.

In the following table, let p_1 and s_1 be the precision and scale of the first operand, and let p_2 and s_2 be the precision and scale of the second operand.

Operation	Precision and Scale of Result	
Addition and Subtraction	Precision:	$\mathrm{MIN}(32, \mathrm{MAX}(p_1 - s_1, p_2 - s_2) + \mathrm{MAX}(s_1, s_2) + 1)$
	Scale:	$\mathrm{MAX}(s_1, s_2)$
Multiplication	Precision:	$\mathrm{MIN}(32, p_1 + p_2)$
	Scale	$s_1 + s_2$
Division	Precision:	32
	Scale:	$32 - p_1 + s_1 - s_2$ (cannot be negative)

Operators and Expressions

Expressions in INFORMIX-4GL can be categorized as numeric, string, and Boolean. Numeric expressions can be either integer (evaluating to INTEGER or SMALLINT) or non-integer (evaluating to FLOAT, SMALLFLOAT, MONEY, or DECIMAL). Because of the automatic conversion capability of INFORMIX-4GL, DATE type variables can occur in either integer or string expressions. Numeric variables can occur in either numeric or string expressions. String variables that are character representations of numbers are converted to numbers when used in numeric expressions. A string variable that is not a character representation of a number will cause an error if used in a numeric expression.

Numeric Expressions

A numeric expression consists of a numeric constant, variable, column name, or function that returns a numeric value, or one of these connected by one of the following arithmetic operators to a numeric expression:

Operator	Operation
**	Exponentiation
*	Multiplication
/	Division
mod	Modulus
+	Addition
–	Subtraction

String Expressions

A string expression is a string constant, a CHAR type variable, a CHAR type column, or a function returning a CHAR type, or combinations of these combined or altered by the following string operators:

Operator	Operation
,	Concatenation
[*m*,*n*]	Substring where *m* is the starting position and *n* is the ending position
USING	Formatting
CLIPPED	Drop trailing blanks

Numeric constants, variables, and columns are converted to their character representation when used in string expressions.

Boolean Expressions

A Boolean expression evaluates to true or false (or unknown if NULL values are involved) and can take any of the following forms:

- *expr rel-op expr*

  Where *expr* is an expression and *rel-op* is a relational operator:

Operator	**Operation**
=	Equal
!= or <>	Not equal
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal

  For string expressions, greater than means after in the ASCII collating order, where lowercase letters are after uppercase letters, which are after numerals. For DATEs, greater than means later in time.

- *string-expr* [NOT] LIKE *string-expr*
- *string-expr* [NOT] MATCHES *string-expr*
- *expr* IS [NOT] NULL
- *expr* [NOT] BETWEEN *expr* AND *expr*
- *expr* [NOT] IN ({*items* | *SELECT-statement*})
- *expr rel-op* {ALL | ANY | SOME} (*SELECT-statement*)
- EXISTS (*SELECT-statement*)

The last four expressions apply only in WHERE clauses of SELECT statements (see Chapter 7 for details).

Boolean expressions may be compounded with the operators NOT, AND, and OR:

[NOT] *Boolean-expr* [{AND | OR} *Boolean-expr*]

Expressions in INFORMIX-4GL Statements

For the CASE, IF, and WHILE statements in INFORMIX-4GL, true is any non-zero number and false is zero. In these statements a numeric expression can be used where a Boolean expression is called for. A Boolean expression can be used where a numeric expression would be expected, yielding 1 or 0. A string expression that is a representation of a number can be used anywhere that a numeric expression is allowed. If you use a string expression where a Boolean expression is expected and the string expression does not represent a number, it will be evaluated as zero or false.

If a Boolean expression contains a NULL value, it has an unknown truth value and, yet, will be treated as false in INFORMIX-4GL statements. This can lead to unfamiliar consequences. If a is a Boolean expression, the compound expression a OR NOT a would, in two-valued logic, be tautologically true. If a contains a NULL value, its truth value is unknown. So is the truth value of NOT a. INFORMIX-4GL treats both as false and their combination with OR as well. If there is any chance that a variable may have a NULL value, you should test it before using it in a Boolean expression. See the section "NULL Values" in Chapter 2 for more details.

Binding to Database and Forms

Regardless of how you have defined them, there is no implicit relationship between program variables, screen fields, and database columns. Even when a variable **lname** is defined to be LIKE **customer.lname**, changes to the program variable do not imply any change in the column value. Similarly, even though you created a screen field **customer.lname** using the same database column as a model, there is no inherent connection between the program variable and the field. You must indicate the binding explicitly in any statement that connects program variables to screen forms or to database columns.

The following two statements take input from the screen and insert the value entered on the screen into the database. The @ sign tells INFORMIX-4GL that the first **lname** is the name of a database column.

```
INPUT lname FROM customer.lname
INSERT INTO customer (@lname) VALUES (lname)
```

Some statements permit temporary binding through the identity of the variable name and the screen field name (see the individual statement descriptions in Chapter 7 for the appropriate syntax). You could use the following statement in place of the first one in the previous example:

```
INPUT BY NAME lname
```

The THRU Keyword and the . Notation*

INFORMIX-4GL provides two devices to simplify the writing of INFORMIX-4GL statements that refer to elements of a record or columns of a table. One of these involves the keyword THRU (or THROUGH), and the other, the .* notation.

```
INITIALIZE pr_rec.element4
    THRU pr_rec.element8 TO NULL

DISPLAY pr_rec.* TO sc_rec.*
```

The first statement sets to NULL the value of the program variables **pr_rec.element4**, **pr_rec.element5**, **pr_rec.element6**, **pr_rec.element7**, and **pr_rec.element8**. The second displays all the values of the record **pr_rec** on the screen in the fields described by the screen record **sc_rec** (see Chapter 3 for a description of screen records).

With one exception, discussed at the end of this section, these devices are "shorthand" for writing out a partial or full list of record elements or the columns of a table with commas separating the individual items. The order of the items in the list is the order they had when defined. There are, however, the following limitations on their use:

- You cannot use THRU in reference to columns of tables. There is no "shorthand" for a partial listing of columns of a table.
- In the definition of a screen record, THRU runs through all the fields in the order they are listed in the Attributes Section of the form specification, from the first named to the last.

```
. . .
ATTRIBUTES
. . .
f002=tab1.aa;
f003=tab1.bb;
f004=tab1.cc;
f005=tab2.aa;
f006=tab2.bb;
f007=tab3.aa;
f008=tab3.bb;
f009=tab3.cc;
. . .
INSTRUCTIONS
SCREEN RECORD sc_rec (tab1.cc THRU tab3.bb)
```

The excerpt above from a form specification file leads to the following list of elements of **sc_rec**:

```
tab1.cc
tab2.aa
tab2.bb
tab3.aa
tab3.bb
```

- You cannot use THRU to indicate a partial list of screen record elements when displaying to a form or inputting from a form.

- You cannot use either THRU or the .* notation on a record that contains an array as elements. For example, you cannot use **myrec.*** to refer to all the elements of a record defined as follows:

```
DEFINE myrec RECORD
    ri   INTEGER,
    ra   ARRAY[2] OF INTEGER
    END RECORD
```

  You can use these "shorthands," however, for records that contain records as elements.

- You may not use THRU or the .* notation in the argument list when defining a function or report. You can, however, list a record as an argument to a function or report.

The exception to .* expanding to a list occurs when you use it in an RDSQL UPDATE statement. The notation

```
UPDATE table1 SET table1.* = pr_rec.*
```

is expanded by INFORMIX-4GL to the proper syntax

```
UPDATE table1
    SET table1.col1 = pr_rec.element1,
        table1.col2 = pr_rec.element2,
        . . .
```

with all SERIAL columns omitted.

INFORMIX-4GL Statements

There are eight different statement types in INFORMIX-4GL that do not deal with the database.

Variable Definition

All program variables must be defined before they can be used.

DEFINE	associates an INFORMIX-4GL identifier with a data type. DEFINE statements must be the first statements within a routine or must be within a GLOBALS statement.

Assignment

You can assign values directly to program variables with one of two statements:

LET	assigns the value of an expression to a simple program variable or to a component of an array or a record. You may not use a single LET statement to assign values to an entire record or an array.
INITIALIZE	assigns default or NULL values to a program variable. You can set default values for all columns in your database (through the **upscol** utility; see Appendix E for details) and use the INITIALIZE statement to assign these values to simple or record variables.

Program Organization

INFORMIX-4GL programs can have three different types of routines: MAIN, FUNCTION, and REPORT. They may also contain global declaration statements.

MAIN	contains one or more INFORMIX-4GL statements and must occur once, and only once, in every INFORMIX-4GL program. INFORMIX-4GL passes program control initially to the MAIN routine when you execute your program. The last statement in the MAIN routine must be the END MAIN statement. After INFORMIX-4GL executes the END MAIN statement, it terminates the program.

FUNCTION	contains a sequence of INFORMIX-4GL statements that perform a desired task and terminates with the END FUNCTION statement. A function can return zero or more values to the routine that called it. You can pass the values of variables to functions as parameters.
REPORT	contains the format and output specification for a report.
GLOBALS	identifies those program variables that have global scope.

There must be one and only one MAIN routine, and there may be only one GLOBALS statement that includes DEFINE statements. There is no restriction on the number of FUNCTION or REPORT segments. You may place all segments in a single system file, put each segment in a separate file, or any combination in between. A system file containing a program segment is called a module.

If there is a GLOBALS statement containing DEFINE statements, it must either be in a module by itself or be the first statement (or the second if there is a DATABASE statement) in the module containing the MAIN segment. Any module containing routines that refer to global variables must begin with a GLOBALS statement giving the pathname of the module that defines the global variables. (See the GLOBALS statement in Chapter 7 for full details.)

Modules may be compiled separately and linked later, permitting you to create a library of INFORMIX-4GL functions and to call them from different modules. In addition, your program can call C functions. See the section "C Functions," later in this chapter, for the rules that C functions must obey if they are called by INFORMIX-4GL programs.

Program Flow

INFORMIX-4GL has many statements that control the flow of a program. These statements are included in INFORMIX-4GL because they simplify the programming process.

CALL	is used to call a function that returns zero or more values. You may only use a function that returns a single value within an expression.
FOR	begins an indexed loop of statements (ended by END FOR) that will be executed until the index reaches a programmer-supplied value.
FOREACH	begins a loop of statements (ended by END FOREACH) that will be executed for each row that is returned by a query of the database.
WHILE	begins a loop of statements (ended by END WHILE) that will be executed until a programmer-supplied Boolean expression becomes false.
CONTINUE	permits a premature cycling of a loop or menu.
EXIT	permits a premature exit from a FOR, FOREACH, WHILE, MENU, INPUT, or CASE statement, or from the entire program.
IF	executes one or more statements conditionally (ended by END IF).
CASE	executes a different sequence of statements (ended by END CASE) depending upon the current value of an expression.
GOTO	passes program control immediately to a designated place in the program.
LABEL	marks the place in the program where a GOTO statement can pass control.

SLEEP	causes the program to pause for a specified length of time.
RUN	executes an operating system program and returns control to the **INFORMIX-4GL** program.

Screen Interaction

The next eighteen statements allow the program to interact with the screen. The first statement constructs a menu, and the next five provide control over clearing the screen, printing messages, retrieving user input, and setting up default values for screen parameters, special keys, and help files. The next three are window management statements. The last nine handle the entry and retrieval of data from screen forms.

MENU	creates a ring menu with help lines, associated help screens, and a description of program behavior for each menu option.
CLEAR	optionally clears particular screen fields, all screen fields, the entire screen, or a window.
ERROR	prints a message in reverse video on the Error Line and rings the terminal bell.
MESSAGE	prints a message on the Message Line.
PROMPT	prints a message on the Prompt Line and returns the user's response.
OPTIONS	specifies the MESSAGE, PROMPT, FORM, and COMMENT lines relative to the screen or current window, as well as any key and help-file designations. You can also specify a new ERROR line with the OPTIONS statement, but the error line remains relative to the screen.

OPEN WINDOW	creates a window, with or without a border, at a given location and makes it available for use. You can explicitly specify the size of the window, or let INFORMIX-4GL determine the size of the window from a given screen form.
CURRENT WINDOW	specifies the current or "topmost" window. All input and output is done in the current window.
CLOSE WINDOW	closes the window that you specify, restoring the "topmost" window (of those that remain) as the current window.
OPEN FORM	opens the compiled screen form and associates an INFORMIX-4GL identifier with the form.
DISPLAY FORM	displays the named form on the screen or in a window, displacing whatever was on the screen or window below the Form Line.
CLOSE FORM	closes the file containing the named screen form and releases its association with the INFORMIX-4GL identifier.
CONSTRUCT	takes user input from a screen form and creates a character string that can be used as the WHERE clause of a SELECT statement. This is the INFORMIX-4GL mechanism for performing a query-by-example.
DISPLAY	displays expressions and variables in fields of a screen form, at a specific position on the screen, in a window, or on the next line.

DISPLAY ARRAY	displays a program array to a screen array and allows scrolling through the array.
INPUT	takes user-entered data from a screen form and puts the data into program variables. You can specify sequences of statements to be executed during the INPUT statement before or after the cursor enters any field or after the user indicates that entry is complete. You can also trap function or control keys and specify an appropriate sequence of statements.
INPUT ARRAY	is an extension of the INPUT statement that takes data entered by a user into a screen array and puts the information into an array of program variables. The user can scroll through the array, insert new rows into the array, and delete rows from the array by using function keys.
SCROLL	moves data in a screen array up or down.

Report Generation

There are three statements in INFORMIX-4GL that control report writing.

START REPORT	signals INFORMIX-4GL to initialize the report. Optionally, you may specify the output device in the START REPORT statement.

OUTPUT TO REPORT	passes a row of the report variables to the report. This statement is usually found within a FOREACH loop where the report row is the current row of a query.
FINISH REPORT	handles the end of report summaries and, if necessary, second passes through the data so that aggregate values can be calculated.

Error and Exception Handling

INFORMIX-4GL allows you to trap runtime errors and warnings and user-entered interrupts (usually DEL or CONTROL-C) and quits (CONTROL-\). The default conditions are that errors, interrupts, and quits cause immediate program termination, while warnings are ignored. You can change these conditions with the following commands:

WHENEVER	allows you to trap errors or warnings and to instruct INFORMIX-4GL to call a function, go to a label, terminate the program, or ignore the problem. In the last case, you must test for errors explicitly after every statement where an error would produce difficulties in your application.
DEFER	allows you to prevent INFORMIX-4GL from terminating a program when the user presses the INTERRUPT or QUIT keys. If you choose this option, the interrupt and/or quit will terminate INPUT, INPUT ARRAY, CONSTRUCT, and PROMPT statements, but will not terminate the program. You must explicitly check the global variables INT_FLAG and QUIT_FLAG to determine whether the user has pressed interrupt or quit after these statements.

Error Handling

INFORMIX-4GL sets the global variable **status** following the execution of every RDSQL or form-related INFORMIX-4GL statement. **status** is zero when the statement executes correctly, negative when there is an error, and equal to NOTFOUND (= 100) when you attempt a FETCH outside the range of the active list of rows or when a SELECT statement can find no rows (see Chapter 2 for more information on the FETCH and SELECT statements). There are three library routines available to the INFORMIX-4GL programmer to identify errors from the **status** variable (see Chapter 5 for a description of library routines).

The WHENEVER statement is designed to trap errors and warnings in the execution of other statements. RDSQL indicates errors and warnings by supplying values for the global record SQLCA. See Chapter 2 for a description of the SQLCA record.

The WHENEVER statement is "shorthand" for writing an IF-THEN statement after every RDSQL or form-related statement that follows the WHENEVER statement in the current source-code module. The CONTINUE option tells INFORMIX-4GL to stop inserting an IF-THEN statement after the subsequent statements. The default is STOP for ERROR and CONTINUE for WARNING. In the default situation, the code generated by INFORMIX-4GL tests for errors, but not warnings, after every INFORMIX-4GL statement.

The scope of a WHENEVER ERROR statement is the module (or file) in which it occurs and from the position of the WHENEVER ERROR statement to the next WHENEVER ERROR statement in the module (or to the end of the module if there are no more WHENEVER ERROR statements). The scope of the WHENEVER WARNING statement is the same.

If you do not specify WHENEVER ERROR CONTINUE, the STARTLOG() function causes the routine name, the error code, and the corresponding error message to be written to a designated error file every time an error occurs. The syntax for STARTLOG() is

```
CALL STARTLOG(filename)
```

where *filename* is a quoted string that is the name of the error log file or a CHAR variable that evaluates to the name of the error log file. If you do not want the error log file to reside in the current directory, you must specify a pathname.

Another library function ERRORLOG(*message)* appends the quoted string or CHAR variable *message* to the error log. This function allows you to write directly to the error log file.

There are three other library routines that deal with errors (see Chapter 5 for a full description of library functions):

ERR_PRINT()	If passed the error code, will print the message on the Error Line.
ERR_GET()	If passed the error code, will return the message to a string variable.
ERR_QUIT()	If passed the error code, will print the message on the Error Line and exit from the program.

Exception Handling

It is also possible to trap interrupts (DEL, CONTROL-C) and quits (CONTROL-\) in your INFORMIX-4GL program. The syntax is:

```
DEFER {INTERRUPT | QUIT}
```

The DEFER keyword means that a global flag (INT_FLAG for INTERRUPT, and QUIT_FLAG for QUIT) is set to non-zero and may be checked by the programmer. It is the programmer's responsibility to reset the flags to zero. The default for either an interrupt or a quit is to cause the program to stop immediately.

If the user enters an interrupt during an INPUT or INPUT ARRAY statement and you have executed the DEFER INTERRUPT statement, the program control leaves the INPUT statement and INFORMIX-4GL sets INT_FLAG.

You may enter the DEFER INTERRUPT statement only once in a program and then only in the MAIN routine. This applies to the DEFER QUIT statement as well.

Data Validation

You can ensure that data entered through a form conforms to appropriate limits or has a permitted value by setting up the INCLUDE attribute in the form specification or in the **syscolval** table (see Chapter 3 for a discussion of these concepts). If your program inserts data into the database from sources other than a form, you can check that it meets these same restrictions by using the VALIDATE statement.

VALIDATE	creates an error if the named variables are not consistent with the limitations set for the named columns in **syscolval**.

Built-in Functions

In addition to functions that you create with the FUNCTION routine and C functions that you can call, INFORMIX-4GL provides a number of built-in functions. You may use the following functions in expressions.

ASCII
CLIPPED
COLUMN
DATE
DATE( )
DAY( )
MDY( )
MONTH( )
SPACES
TODAY
USING
WEEKDAY( )
YEAR( )

These functions are described in the following sections. There are additional functions that can be used only within REPORT routines (see Chapter 4 for details).

ASCII

Overview

INFORMIX-4GL evaluates this function as a single character. You can use it to display CONTROL characters.

Syntax

```
ASCII num-expr
```

Explanation

ASCII	is a required keyword.
num-expr	is a numeric expression.

Examples

The following DISPLAY statement rings the terminal bell (ASCII value of 7).

```
DEFINE bell CHAR(1)
LET bell = ASCII 7
DISPLAY bell
```

The next REPORT routine fragments show how to implement special printer or terminal functions. They assume that, when the printer receives the sequence of ASCII characters 9, 11, and 1, it will start printing in red and, when it receives 9, 11, and 0, it will revert 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.

```
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
      . . .
```

Caution! INFORMIX-4GL cannot distinguish printable and non-printable ASCII characters. Be sure to account for the non-printing characters when using the COLUMN function to format your page. Since various devices differ in outputting spaces with CONTROL characters, you may have to use trial and error to line up columns when you output CONTROL characters.

CLIPPED

Overview

CLIPPED displays the character expression that precedes it without any trailing blanks.

Syntax

---

char-expr CLIPPED

Explanation

char-expr	is a required character expression.
CLIPPED	is a required keyword.

Notes

1. You normally use CLIPPED following a variable name in a LET, DISPLAY or PRINT (REPORT routine only) statement.

Examples

The following example is from a REPORT routine that prints mailing labels.

```
FORMAT
  ON EVERY ROW
    IF (city IS NOT NULL) AND (state IS NOT NULL) THEN
       PRINT fname CLIPPED, 1 SPACE, lname
       PRINT company
       PRINT address1
       IF (address2 IS NOT NULL) THEN
          PRINT address2
    END IF
       PRINT city CLIPPED, ", " , state, 2 SPACES, zipcode
       SKIP TO TOP OF PAGE
  END IF
```

COLUMN

Overview

The COLUMN function returns a string of spaces long enough to begin the next item in the designated column.

Syntax

```
COLUMN integer-expr
```

Explanation

COLUMN	is a required keyword.
integer-expr	is a required positive integer expression that specifies the initial column number of the next item to be printed.

Notes

1. In REPORT routines, **INFORMIX-4GL** calculates the column number by adding the number to the left margin you set in the OUTPUT section. Otherwise, the column number is counted from the left edge of your screen.

2. If **INFORMIX-4GL** has already printed past the column specified by *integer-expr*, **INFORMIX-4GL** ignores the COLUMN expression.

3. If you use the COLUMN function in a DISPLAY statement, you must specify an integer instead of an integer expression after the COLUMN keyword.

Examples

```
PAGE HEADER
  PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35, "CITY",
    COLUMN 57, "ZIP", COLUMN 65, "PHONE"
  SKIP 1 LINE
ON EVERY ROW
  PRINT customer_num, COLUMN 12, fname CLIPPED, 1 SPACE,
    lname CLIPPED, COLUMN 35, city CLIPPED, ", ", state,
    COLUMN 57, zipcode, COLUMN 65, phone
```

DATE

Overview

The DATE function returns a character string with the value of today's date in the format "Thu Sep 20 1986."

Syntax

DATE

Explanation

DATE	is a required keyword.

Examples

Because DATE returns type CHAR, you can use it with subscripts to express a day, month, date, or year. The following example displays the three-letter abbreviation for the day of the week.

```
DISPLAY "Today is ", DATE[1,3]

          Today is Mon
```

DATE()

Overview

The DATE() function returns a type DATE value corresponding to the expression with which you call it.

Syntax

```
DATE(expr)
```

Explanation

DATE	is a required keyword.
expr	is a required expression that can be converted to a type DATE value.

Examples

The following example uses DATE to convert a string to a date.

```
    WHERE end_date > DATE("12/13/1986")
```

DATE(100) returns the 100th day after December 31, 1899.

DAY()

Overview

The DAY() function returns the day of the month when you call it with a type DATE expression.

Syntax

```
DAY(date-exp)
```

Explanation

DAY	is a required keyword.
date-exp	is a required expression of type DATE.

MDY()

Overview

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.

Syntax

MDY(*expr1*, *expr2*, *expr3*)

Explanation

MDY	is a required keyword.
expr1	is an expression that evaluates to an integer representing the number of the month (1–12).
expr2	is an expression that evaluates to an integer representing the number of the day of the month (1–28, 29, 30, or 31, depending on the month).
expr3	is an expression that evaluates to a four-digit integer representing the year.

Notes

1. The value of *expr3* cannot be the abbreviation for the year. For example, 84 is in the first century.

MONTH()

Overview

The MONTH() function returns an integer corresponding to its type DATE argument.

Syntax

```
MONTH(date-expr)
```

Explanation

MONTH	is a required keyword.
date-expr	is a required expression of type DATE.

SPACES

Overview

This function is used only within reports to return a string of spaces. It is identical to a quoted string of spaces.

Syntax

---

num-expr SPACE[S]

Explanation

num-expr	is a numeric expression.
SPACES	is a required keyword. You can use the keyword SPACE in place of SPACES if you like.

Notes

1. You can only use the SPACES function in PRINT statements that appear in REPORT routines. (See Chapter 4 for more information about PRINT.)

Examples

The following example is from a mailing label report.

```
FORMAT
  ON EVERY ROW
      PRINT fname, lname
      PRINT company
      PRINT address1
      PRINT address2
      PRINT city, ", " , state, 2 SPACES, zipcode
      SKIP 2 LINES
```

TODAY

Overview

TODAY evaluates as type DATE with a value of today's date as supplied by the operating system.

Syntax

```
TODAY
```

Explanation

TODAY	is a required keyword.

Examples

The following example is from a REPORT routine.

```
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

Overview

The USING function allows you to format a numeric or date expression. With a numeric 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 will convert the date to a variety of formats.

Syntax

---

expr1 USING *expr2*

Explanation

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*. Refer to "Notes" later in this section.

Formatting Numeric Expressions. The format string consists of combinations of the following characters: * & # < , . - + ( ) $. The characters - + ( ) $ will *float.* When a character floats, INFORMIX-4GL 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 for an explanation of these characters.

* This character fills with asterisks any positions in the display field that would otherwise be blank.

& This character fills with zeros any positions in the display field that would otherwise be blank.

This character does not change any blank positions in the display field. You can 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 literal; USING displays it as a comma. USING will not display a comma unless there is a number to the left of it.

. This character is a literal; USING displays it as a period. You can only have one period in a format string.

- This character is a literal; USING displays it as a minus sign when *expr1* is less than zero. When you group several minus signs in a row, a single minus sign floats to the right-most 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 plus signs 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 parentheses 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. One of these characters generally closes a format string that begins with a left parenthesis.

$ This character is a literal; USING displays it as a dollar sign. When you group several dollar signs in a row, a single dollar sign floats to the rightmost position without interfering with the number being printed.

Refer to the "Examples" section for examples of formatting numeric expressions.

Formatting Date Expressions. The format string consists of combinations of the characters *m*, *d*, and *y*, as shown in Figure 1-1. Refer to the "Examples" section for examples of formatting date expressions.

dd	day of the month as a 2-digit number (01-31)
ddd	day of the week as a 3-letter abbreviation (Sun through Sat)
mm	month as a 2-digit number (01-12)
mmm	month as a 3-letter abbreviation (Jan through Dec)
yy	year as a 2-digit number in the 1900s (00-99)
yyyy	year as a 4-digit number (0001-9999)

Figure 1-1. Combinations of Date Format Strings

Notes

1. The format string must appear between quotation marks.

2. Although USING is generally used as part of a DISPLAY or PRINT statement, you can also use it with LET.

3. If you attempt to display a number that is too large for a display field, INFORMIX-4GL will fill the field with asterisks to indicate an overflow.

Examples

The example below prints the balance field using a format string that allows up to $9,999,999.99 to be formatted correctly.

```
DISPLAY "The current balance is ",
      23485.23 USING "$#,###,##&.&&"
```

The result of executing this DISPLAY statement with the value 23,485.23 is shown below.

```
The current balance is $    23,485.23
```

This example fixes the dollar sign. If dollar signs were used instead of # characters, the dollar sign would have floated with the size of the number. The example 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 &s will appear as zeros, not blanks.

The following example is from a REPORT routine.

```
ON LAST ROW
   SKIP 2 LINES
   PRINT "Number of customers in ", state, " are ",
         COUNT(*) USING "<<<<<"

PAGE TRAILER
   PRINT COLUMN 35, "page ", PAGENO USING "<<<<"
```

The following example is from a complex REPORT routine and illustrates several different formats.

```
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 number
   PRINT COLUMN 2, order_date, COLUMN 15, company CLIPPED,
         COLUMN 35, fname CLIPPED, 1 SPACE, lname CLIPPED,
         COLUMN 55, number USING "####",
         COLUMN 60, GROUP SUM (total_price)
              USING "$$,$$$,$$$.&&"

AFTER GROUP OF days
   SKIP 1 LINE
   PRINT COLUMN 21, "Total amount ordered for the day:  ",
         GROUP SUM (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:  ", SUM (total_price)
         USING "$$$$,$$$,$$$.&&"

PAGE TRAILER
   PRINT COLUMN 28, PAGENO USING "page <<<<"
```

The displays on the next four pages illustrate the full power of the USING function.

Format String	Numeric Value	Formatted Result
"#####"	0	bbbbb
"&&&&&"	0	00000
"$$$$$"	0	bbbb$
"*****"	0	*****
"<<<<<"	0	(null string)
"##,###"	12345	12,345
"##,###"	1234	b1,234
"##,###"	123	bbb123
"##,###"	12	bbbb12
"##,###"	1	bbbbb1
"##,###"	−1	bbbbb1
"##,###"	0	bbbbbb
"&&,&&&"	12345	12,345
"&&,&&&"	1234	01,234
"&&,&&&"	123	000123
"&&,&&&"	12	000012
"&&,&&&"	1	000001
"&&,&&&"	0	000000
"$$,$$$"	12345	****** (overflow)
"$$,$$$"	1234	$1,234
"$$,$$$"	123	bb$123
"$$,$$$"	12	bbb$12
"$$,$$$"	1	bbbb$1
"$$,$$$"	0	bbbbb$
"**,***"	12345	12,345
"**,***"	1234	*1,234
"**,***"	123	***123
"**,***"	12	****12
"**,***"	1	*****1
"**,***"	0	******

This table uses the character b to represent a blank or space.

Format String	Numeric Value	Formatted Result
"##,###.##"	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
"&&,&&&.&&"	12345.67	12,345.67
"&&,&&&.&&"	1234.56	01,234.56
"&&,&&&.&&"	123.45	000123.45
"&&,&&&.&&"	0.01	000000.01
"$$,$$$.$$"	12345.67	********* (overflow)
"$$,$$$.$$"	1234.56	$1,234.56
"$$,$$$.##"	0.00	$.00
"$$,$$$.##"	1234.00	$1,234.00
"$$,$$$.&&"	0.00	$.00
"$$,$$$.&&"	1234.00	$1,234.00
"-##,###.##"	-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
"---,###.##"	12.34	12.34
"---,-##.##"	12.34	12.34
"---,---.##"	1.00	1.00
"---,---.--"	-.01	-.01

This table uses the character b to represent a blank or space.

Format String	**Numeric Value**	**Formatted Result**
"————,———.&&"	−.01	−.01
"—$$$,$$$.&&"	−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	−$12,345.67
"————,——$.&&"	−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	($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)

Format String	Numeric Value	Formatted Result
"($$$,$$$.&&)"	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
"<<<,<<<"	12345	12,345
"<<<,<<<"	1234	1,234
"<<<,<<<"	123	123
"<<<,<<<"	12	12

The following examples show sample conversions for December 25, 1986.

Format String	Formatted Result
"mmddyy"	122586
"ddmmyy"	251286
"yymmdd"	861225
"yy/mm/dd"	86/12/25
"yy mm dd"	86 12 25
"yy—mm—dd"	86—12—25
"mmm. dd, yyyy"	Dec. 25, 1986
"mmm dd yyyy"	Dec 25 1986
"yyyy dd mm"	1986 25 12
"mmm dd yyyy"	Dec 25 1986
"ddd, mmm. dd, yyyy"	Thu, Dec. 25, 1986
"(ddd) mmm. dd, yyyy"	(Thu) Dec. 25, 1986

WEEKDAY()

Overview

The WEEKDAY() function returns an integer that represents the day of the week when you call it with a type DATE expression.

Syntax

```
WEEKDAY(date-expr)
```

Explanation

WEEKDAY	is a required keyword.
date-expr	is a required expression of type DATE.

Notes

1. WEEKDAY returns an integer in the range 0-6.
   Zero represents Sunday, 1 represents Monday, and so on.

YEAR()

Overview

The YEAR() function returns an integer that represents the year (four digits for 1986) when you call it with a type DATE expression.

Syntax

```
YEAR(date-expr)
```

Explanation

YEAR	is a required keyword.
date-expr	is a required expression of type DATE.

C Functions

INFORMIX-4GL programs can call C language functions. To provide a transparent function calling mechanism, INFORMIX-4GL requires that C functions follow a *calling convention* that allows data to be passed between the INFORMIX-4GL program and the C function.

The convention utilizes a stack, which is a data structure that can be accessed in a pre-defined way by both INFORMIX-4GL programs and C functions. The operations on a stack are either to add a variable to the stack, "push," or to retrieve a variable from the stack, "pop." The stack acts as a Last-In First-Out queue, so that the last variable added to the stack is the next variable to be removed from the stack if you do a pop.

Consider the following INFORMIX-4GL statement:

```
    CALL myfunc (a,b,c)
```

Part of the calling convention is that function arguments are pushed from left to right onto the stack. INFORMIX-4GL automatically pushes the variables **a**, **b**, and **c** onto the stack, in that order. Another part of the calling convention requires INFORMIX-4GL to automatically pass the number of calling parameters as the only real argument to the function.

The C function designed to work with **INFORMIX-4GL** must obey the calling convention by complementing the actions of **INFORMIX-4GL**. All compatible C functions have only one argument, which is the number of parameters that **INFORMIX-4GL** placed on the stack. You use this argument to pop the calling parameters. **INFORMIX-4GL** supplies a number of library functions to assist this process. The library functions are identified as follows:

Data Definition		**Popping Functions**	**Pushing Functions**
int	i;	popint(&i)	retint(i)
short	s;	popshort(&s)	retshort(s)
long	l;	poplong(&l)	retlong(l)
float	f;	popflo(&f)	retflo(&f)
double	d;	popdub(&d)	retdub(&d)
char	str[m];	popquote(str, m)	retquote(str)
dec_t	dm;	popdec(&dm)	retdec(&dm)

Your first step in a C function is to pop all the arguments in the reverse order from the order in the call. If the call is to **myfunc**, you must first pop **c**, and then **b**, and, finally, **a**. You must use the library function that is appropriate for the data type of the calling argument. Unless you pop all the arguments, the stack becomes corrupted, and your **INFORMIX-4GL** program can produce unpredictable results.

To return values from the C function to the **INFORMIX-4GL** program, you must push the values onto the stack. They must be pushed in the same order as they appear in the RETURNING clause of your **INFORMIX-4GL** statement. For example, if the statement is

```
CALL ... RETURNING x, y
```

you must push **x** before pushing **y** within your C function. You must use the library function that matches the data type to execute the push. The last statement that your C function executes must be a **return** where the only parameter is the number of variables that are being returned. Make sure that the variables returned by your C function match, in both number and data type, the arguments in the RETURNING clause of your INFORMIX-4GL statement. If you do not return the correct data types, your program can execute unpredictably. Failure to return the expected number of arguments causes an error.

When using the C library functions, you must be aware of the following factors:

- If you use the **retquote(str)** function, you must null-terminate the string.
- The string **str** in **popquote** will be null-terminated; you should allow for that in the value of **m**.
- The **dec__t** structure is defined in Appendix K, along with a number of useful functions that you can use to convert DECIMAL variables to other numeric data types and back again within your C functions. (They are not necessary within an INFORMIX-4GL program since the functionality is already built into INFORMIX-4GL.)

Examples

The first example shows the basic structure for C functions that you call from an INFORMIX-4GL program.

```
myfunc(n)
int n;
{
    /* n specifies how many 4gl parameters
     *    came in
     */

    test that the value of n is correct

    pop n 4gl parameters in reverse order,
                            right to left

    ... code

    push x return 4gl parameters left to right

    return(x)
}   /* must return the number x of 4gl
     *      "RETURNING"
     */
```

The following INFORMIX-4GL program CALLs the C language function **sndmsg**, which converts a string into EBCDIC and sends it to a remote computer. Two arguments are explicitly passed to the function: the source string and its length. INFORMIX-4GL automatically passes the number of arguments to the function.

```
MAIN

   DEFINE chartype CHAR(80),
          msg_status INTEGER,
          return_code INTEGER

   LET chartype = "234"

#  sndmsg requires two arguments and returns two arguments,
#  as defined by the C language function.

#  You must ensure that the order and data types of all
#  arguments are compatible between the 4gl calling
#  program and the called function.

   CALL sndmsg(chartype, 4) RETURNING msg_status, return_code

   IF return_code <> 0 THEN

      DISPLAY "Error code: ", return_code

   END IF

   DISPLAY msg_status

END MAIN
```

The function **sndmsg** checks that the correct number of arguments are passed. **INFORMIX-4GL** cannot guarantee recovery of the stack if a function is called with the wrong number of arguments, since that is a failure to follow the calling convention between **INFORMIX-4GL** and C functions. **sndmsg** aborts the program if the wrong number of arguments is passed.

If the correct number of arguments is passed, **sndmsg** pops the arguments from the stack, using the library functions appropriate for the data type.

```
#include "stdio.h"
#include "decimal.h"
sndmsg(nargs)

/* 4gl syntax is CALL sndmsg (input,len) RETURNING msg_number, retcode */

int nargs;  /* 4gl passes the number of arguments as an integer */

{

char    input[80];        /* 4gl and C function must agree on data  */
int     msg_number;       /* types and the order that arguments are */
int     len, retcode;     /* placed on the stack                    */

/* Check that the correct number of arguments are passed */

   if (nargs != 2)
      {
      fprintf (stderr,
              "sndmsg:  wrong number of arguments");
      exit(1); /* No recovery from this error */
      }

/* Pop rightmost argument */

   popint(&len);

/* Pop next argument      */

   popquote(input,len);

/* Finished with function calling convention */
/* Start function processing                 */

   msg_number =-1;
   retcode = cvtebcd (&input, len);     /* user-written function */
   if (retcode != 0)
      msg_number = sndrmt (input,len); /* user-written function */

/* Finished processing */
/* Return (push) leftmost argument */

   retint(msg_number);

/* Return next argument */

   retint(retcode);

/* Finished returning arguments */
/* Return from function giving number of arguments */

   return(2);

}
```

To return values to the INFORMIX-4GL program, the C function uses the appropriate push functions for the data type and pushes return arguments onto the stack from left to right. The last statement executed by the function returns the number of arguments, which is two for **sndmsg**. A discrepancy between the number of return arguments in the function and the number of arguments expected by the INFORMIX-4GL program results in an error.

Compiling and Executing Programs

This section describes how to compile your INFORMIX-4GL program from the operating system level. You may prefer to write and compile your program within the INFORMIX-4GL Programmer's Environment. (See Chapter 6 for a description of the Programmer's Environment.)

To convert the INFORMIX-4GL code that you write into an executable program, four steps are necessary:

1. Preprocess the INFORMIX-4GL code to produce INFORMIX-ESQL/C code.

2. Preprocess the INFORMIX-ESQL/C code to produce C language code.

3. Compile the C code with the C compiler to create an object file.

4. Link the object file with the INFORMIX-ESQL/C libraries, as well as your own.

You can use the **c4gl** command file that is installed with INFORMIX-4GL to perform all these tasks.

To preprocess and compile an INFORMIX-4GL program, give the source module name the extension **.4gl** and enter the following command in response to your system prompt:

```
c4gl [-e][-a][-otherargs ...][-o outfile] \
     source.4gl ... [otheresql.ec ...] \
     [othersrc.c ...] \
     [otherobj.o ...][-lyourlib ...]
```

Explanation

-e	allows you to request just the preprocessor steps with no compilation or linking.

–a	causes your compiled program to check array bounds at runtime. The **–a** option must appear on the command line before the **source.4gl** filename.
otherargs	are other arguments that you want to pass to the C compiler.
outfile	is the name of the executable output file.
source.4gl	is the name of your INFORMIX-4GL source module.
otheresql.ec	is an INFORMIX-ESQL/C source file that you want to compile and link.
othersrc.c	is a C language source file that you want to compile and link.
otherobj.o	is an object file that you want to link with your INFORMIX-4GL program.
yourlib	is a library that you want to extract functions from (for example, **libm.a** for mathematical functions).

Notes

1. If you are using a DOS system on MS-NET, you will not be able to compile INFORMIX-4GL programs, link files, or use a **c4gl** command file while you are on the net. This is due to the memory requirements of the MS-NET software. To compile an INFORMIX-4GL program, you need to have available 640K of memory.

   To ensure that you have sufficient memory available to compile your programs, first exit the network. Then compile and link the programs from within the Programmer's Environment or using **c4gl**. You can then reenter the network to run your programs.

2. Since the **–a** option requires additional processing, you may want to use this option only during development for debugging purposes.

3. **c4gl** passes all C arguments (**–otherargs**) and other C source and object files (**othersrc.c** and **otherobj.o**) directly to the C compiler **cc**.

4. If there is no MAIN statement in **source.4gl**, your code will be compiled to **source.o**, but will not be linked with other routines or the libraries. You can link it with a main routine at another time. In this way, you can compile INFORMIX-4GL modules separately from your MAIN module.

Examples

The simplest case is to compile a single-module INFORMIX-4GL program:

```
c4gl single.4gl –o single
```

If **mod1.o**, **mod2.o**, and **mod3.o** are previously compiled INFORMIX-4GL modules and you want to compile and link **mod4.4gl** to create the executable program **myappl**, use the following command line:

```
c4gl mod1.o mod2.o mod3.o mod4.4gl –o myappl
```

Program Filename Extensions

The source, executable, object, error, and backup files generated by **INFORMIX-4GL** are stored in the current directory and are labeled with a filename extension. The list below shows the file extensions for the source, executable, object, and error files. These files are produced during the normal course of using **INFORMIX-4GL**. The backup files appear in a later list.

file.4gl	is an **INFORMIX-4GL** source file.
file.o	is an **INFORMIX-4GL** object file.
file.4ge	is an **INFORMIX-4GL** executable (runable) file.
file.err	is an **INFORMIX-4GL** source error file, created when an attempt to compile a module or form specification fails. The file contains **INFORMIX-4GL** source code plus compiler syntax errors.
file.ec	is an intermediate source file, created during the normal course of compiling an **INFORMIX-4GL** module.
file.c	is an intermediate C file, created during the normal course of compiling an **INFORMIX-4GL** module.
form.per	is a **FORM4GL** form source file.
form.frm	is a **FORM4GL** form object file.

The following list contains the **INFORMIX-4GL** backup files:

file.4bl	is an **INFORMIX-4GL** source backup file, created during the modification and compilation of a program module.
file.4bo	is an **INFORMIX-4GL** object backup file, created during the modification and compilation of a program module.

file.4be	is an INFORMIX-4GL executable backup file, created during the modification and compilation of a program module.
file.erc	is an INFORMIX-4GL object and/or executable error file, created when an attempt to compile a non-INFORMIX-4GL module fails or during the linking phase of the compilation. The file contains INFORMIX-4GL source code plus annotated compiler errors.
file.pbr	is a FORM4GL form source backup file.
file.fbm	is a FORM4GL form object backup file.

Under normal conditions, INFORMIX-4GL creates the backup and intermediate files as necessary and deletes them upon a successful compilation. If you interrupt a compilation, you may find one or more of the files in your current directory.

The **file.4bl** source backup file contains a copy of the **file.4gl** source file. During the compilation process, INFORMIX-4GL modifies the (original) **file.4gl** source file. In the event of a system crash, you may need to replace the modified **file.4gl** file with the backup copy contained in the **file.4bl** file.

Chapter 2

Using RDSQL

Chapter 2 Table of Contents

Chapter Overview 5

Relational Databases 6

RDSQL Identifiers 6

Database Data Types 7

RDSQL Statement Summary 10

Data Definition 10

Data Manipulation 13

Cursor Management 14
- SELECT Cursors 14
  - Associating a Cursor with a SELECT Statement 15
  - Retrieving and Processing Rows 16
  - Deleting or Updating the Current Row 22
- INSERT Cursors 26

Dynamic Management 30
- Preparing Statements 31
  - Statements That Require No Input 32
  - Statements That Require Input for Values 32
  - Statements That Require Input for RDSQL Identifiers 34
- Executing Prepared Statements 35
  - The EXECUTE Statement 36
  - Running Prepared SELECT Statements 38
  - Running Prepared INSERT Statements 40

Data Access 42

Data Integrity 43
- Transactions 43
- Transaction Log File Maintenance 46
- Audit Trails 47
  - Creating an Audit Trail 47

Recovering a Table 48
Comparison of Transactions and Audit Trails 48

User Status and Privileges 49

Indexing Strategy 51
Auto-Indexing 53
Clustered Indexes 53

Locking 54
Row-Level Locking 55
Row-Level Locking in Transactions 56
Table-Level Locking 56
Wait for Locked Row 57

NULL Values 58
Default Values 58
The NULL in Expressions 59
The NULL in Boolean Expressions 60
The NULL in WHERE Clauses 61
The NULL in ORDER BY Clauses 62
The NULL in GROUP BY Clauses 62
The NULL Keyword in INSERT and UPDATE Statements 63

Views 64
Creating and Deleting Views 65
Querying Through Views 66
Modifying Through Views 66
Privileges with Views 68
Data Constraints Using Views 68

Outer Joins 70

Table Access by Row ID 71

SQLCA Record 72

TODAY and USER Functions 75

Chapter Overview

Relational Database Systems, Inc., (RDS) has developed **RDSQL** as an extension of the Structured Query Language (SQL) developed by IBM. The RDS additions to the language permit you to change databases, change the names of tables and columns, and increase the functionality of ANSI standard SQL statements. In the family of RDS database products, **RDSQL** plays several roles. In **INFORMIX-SQL**, **RDSQL** is both the interactive query language and the language you use to choose the data for **ACE**, the **INFORMIX-SQL** report-writing program. Read about these uses of **RDSQL** in the *INFORMIX-SQL User Manual.* In **INFORMIX-ESQL/C**, **RDSQL** is the database query language that you embed in C programs to create an application. In **INFORMIX-4GL**, **RDSQL** statements are combined with those described in Chapter 1 to form an almost seamless fourth-generation language.

This chapter describes **RDSQL** and gives an overview of its statements. The full syntax and rules governing **RDSQL** statements are located in Chapter 7 of the *INFORMIX-4GL Reference Manual.*

Relational Databases

The statements of RDSQL create and manipulate *relational databases.* A relational database consists of one or more *tables* that, in turn, are constructed of *rows* and *columns.* Each row contains a particular set of column values. Databases are created as subdirectories of the current directory. The name of the directory is the database name with the extension **.dbs**. The database subdirectory contains nine *system catalog* tables that define the database dictionary. It also contains the tables that constitute the database. Each of these tables is represented by data files and index files with the extensions **.dat** and **.idx**, respectively. The system catalogs are described in Appendix B.

RDSQL Identifiers

An RDSQL *identifier* is the name of an object and can consist of letters, numbers, and underscores (_). The first character must be a letter. Unless qualified to the contrary, an identifier can have from one to eighteen characters.

Database	A database name is an identifier that can have from one to ten (UNIX) or eight (DOS) characters.
Table	A table name is an identifier that must be unique within the database.
Column	A column name is an identifier that must be unique within a table; there can be duplicate column names within a database. When column names within different tables are not unique, use the notation *table.column* to specify the intended column. If you intend to define an INFORMIX-4GL record like a table, the first eight characters of each column name in the table must be unique within the table.

If there is an ambiguity because an **INFORMIX-4GL** identifier and an **RDSQL** identifier are the same, **INFORMIX-4GL** assumes that the **INFORMIX-4GL** identifier refers to a program variable and not to the **RDSQL** object. If you want to override this default assignment, prefix the **RDSQL** identifier with an @ sign. For example, if **lname** is defined as a program variable and you wish to refer to the database column of the same name, use **@lname** for the column name:

```
SELECT @lname INTO lname FROM customer
```

Database Data Types

You must assign a data type to every column in the database (see the CREATE TABLE statement in Chapter 7). With the exception of the SERIAL data type, the definitions of the data types here are identical with the definitions of the same types in Chapter 1. Following are the valid **RDSQL** data types:

CHAR(*n*)	is a character string of length *n* (where $1 <= n <= 32,767$).
SMALLINT	is a whole number from −32,767 to +32,767.
INTEGER	is a whole number from −2,147,483,647 to +2,147,483,647.
DECIMAL[(*m*[,*n*])]	is a decimal floating-point number with a total of *m* (<= 32) significant digits (the precision) and *n* (<= *m*) digits to the right of the decimal point (the scale). When you give values for both *m* and *n*, the decimal variable has fixed-point arithmetic. All numbers less than 0.5×10^{-n} in absolute value have the value zero. The largest absolute value of a variable of this type that can be stored without an error is $10^{m-n} - 10^{-n}$.

	The second parameter is optional and, if missing, the variable is treated as a floating decimal. This means that DECIMAL(*m*) variables have a precision of *m* and a range in absolute value from 10^{-128} to 10^{126}. If no parameters are designated, DECIMAL is treated as DECIMAL(16), a floating decimal.
SMALLFLOAT	is a binary floating-point number corresponding to the single-precision, floating-point data type. The range of values for a SMALLFLOAT data type is the same as the range of values for the “float” data type in the C language on your machine. On some small machines, **RDSQL** implements SMALLFLOAT as DECIMAL(8). See the “Small Machine Features” note that follows.
FLOAT	is a binary floating-point number corresponding to the double-precision, floating-point data type. The range of values for a FLOAT data type is the same as the range of values for the “double” data type in the C language on your machine. On some small machines, **RDSQL** implements FLOAT as DECIMAL(16). See the “Small Machine Features” note that follows.
MONEY[(*m*[,*n*])]	can also take two parameters like the DECIMAL data type. The limitation on values for columns of type MONEY(*m*, *n*) is the same as for columns of type DECIMAL(*m*, *n*). The type MONEY(*m*) is defined as DECIMAL(*m*, 2) and, if no parameter is given, MONEY is taken to be DECIMAL(16, 2). Regardless of the

	number of parameters, the data type MONEY is always treated as a fixed decimal number.
SERIAL[(*n*)]	is a unique sequential integer assigned automatically by **RDSQL**. You can assign an initial value *n*. The default starting integer is 1.
DATE	is a date entered as a character string in one of the formats described in Chapter 1 and stored as an integer number of days since December 31, 1899.

Small Machine Features: On some small machines, SMALLFLOAT and FLOAT columns are implemented as DECIMAL(8) and DECIMAL(16), respectively. If you are not sure whether your computer falls into the "small machine" category, you can create a table with a SMALLFLOAT or FLOAT column and then query the **syscolumns** table for the data type of the column. (See the appendix "System Catalogs" for more information about the **syscolumns** table.)

If you have a small machine, the automatic conversion of SMALLFLOAT and FLOAT data types to DECIMAL data type occurs when you create or modify a table. Although **INFORMIX-4GL** does not change the data definition in your CREATE TABLE or ALTER TABLE statement, it records the column data type as DECIMAL in the database, treats SMALLFLOAT and FLOAT numbers as DECIMAL numbers, and stores column data as DECIMAL(8) or DECIMAL(16). You may find it easier to define SMALLFLOAT and FLOAT columns as DECIMAL columns so that your CREATE TABLE statement reflects the table structure stored in the database.

RDSQL Statement Summary

Six different types of RDSQL statements are used with INFORMIX-4GL:

- Data definition
- Data manipulation
- Data access
- Cursor management
- Data integrity
- Dynamic management

Data Definition

Data definition statements include those that create and drop a database and its tables, views, and indexes, modify tables, indexes, and columns, or rename tables and columns. Of this list, only the DATABASE statement is required before manipulating the data of an existing database or defining program variables LIKE columns in the database.

CREATE DATABASE	creates a database directory, sets up the system catalogs, and makes the new database the *current database.* There can be no more than one current database at any time.
DATABASE	selects a database and makes it the current database. There can be no more than one current database at any time.
CLOSE DATABASE	closes the current database files and leaves no database current. The only RDSQL statements permitted when there is no current database are: ● CREATE DATABASE ● DATABASE ● DROP DATABASE

- START DATABASE

DROP DATABASE	deletes all tables, indexes, and system catalogs. If no other files are present in the database subdirectory, the subdirectory is also deleted.
CREATE TABLE	creates a table and defines the columns and their data types.
ALTER TABLE	adds and drops columns from a table and modifies data types of existing columns.
RENAME TABLE	changes the name of a table.
DROP TABLE	deletes all data and indexes for a table and erases its entry in the system catalogs.
CREATE VIEW	defines a table selected from rows and columns of existing tables and views. As the underlying tables change, so does the view built upon them. See the section "Views" later in this chapter for more information about views.
DROP VIEW	deletes the definition of the view from the system catalogs along with any views defined in terms of the one that is dropped. The underlying tables are unaffected.

CREATE SYNONYM	defines an alternative name for a table or a view. A synonym is effective only for the user who creates it and can be dropped only by its creator. This means that if several people want to use the same synonym, they must each create the synonym. For **INFORMIX-4GL** programs, the creator is the user who runs the program that creates the synonym.
DROP SYNONYM	deletes a synonym from the system catalogs.
RENAME COLUMN	changes the name of a column.
CREATE INDEX	creates an index on one or more columns of a table. See the section "Indexing Strategy" later in this chapter for more information on indexes.
ALTER INDEX	clusters a table in the order of an existing index or releases an index from the cluster attribute.
DROP INDEX	deletes a previously CREATEd index.
UPDATE STATISTICS	updates the system catalogs by determining and inserting the number of rows in the indicated tables. **RDSQL** uses this information in optimizing queries, but does not automatically update the system catalogs after each INSERT or DELETE.

Data Manipulation

The data manipulation statements are the most frequently used RDSQL statements:

DELETE	deletes one or more rows from a table.
INSERT	adds one or more rows to a table.
SELECT	retrieves data from one or more tables.
UPDATE	modifies the data in one or more rows of a table.

SELECT is the most important and the most complex RDSQL statement. Although its syntax is defined in detail in Chapter 7, the following examples illustrate its use.

```
SELECT lname, company
   INTO p_lname, p_company
   FROM customer
   WHERE customer_num = 101
```

This statement queries the **customer** table and returns the single row for which the customer number is 101. From that row, it selects and places the values in the columns corresponding to the contact's last name and company name in the program variables **p_lname** and **p_company**.

```
SELECT @quantity, @total_price
   INTO quantity, total_price
   FROM items
   WHERE order_num = 1001
```

This example shows another SELECT statement that returns a single row. In this example, the program variables **quantity** and **total_price** have the same identifier as the corresponding columns in the **items** table. There is no conflict here since the prefixed @ distinguishes the column name from the program variable.

A SELECT statement that returns a single row is called a *singleton* SELECT statement and can stand alone. The section "Cursor Management" that follows describes how to handle SELECT statements that return more than one row.

Cursor Management

The following sections describe how to use a *cursor* to handle SELECT statements that return more than one row, or to insert rows into the database as a block.

SELECT Cursors

In the examples of a previous subsection, the SELECT statement returned exactly one row, and the values returned to the program variables are unambiguous. When more than one row can be returned, it is necessary to have a device to distinguish one row from another. This device is called a *cursor.*

The set of rows returned by a SELECT statement is called the *active set* for the statement. Within an **INFORMIX-4GL** program, you can work with only one row of the active set at a time. This row is called the *current row* and is referenced by a cursor.

A cursor can be in one of two states: open or closed. When a SELECT cursor is in an open state, it is associated with an active set and can point to the current row, between two rows, before the first row, or after the last row. When it is in a closed state, the cursor no longer is associated with an active set, although it remains associated with the SELECT statement.

The following sections describe how to use the cursor management statements to process rows returned by a

SELECT statement. (For complete information on the syntax of each statement, see Chapter 7 of the *INFORMIX-4GL Reference Manual.*)

Associating a Cursor with a SELECT Statement

You can use the DECLARE statement to name a cursor and associate it with a SELECT statement. You can DECLARE a non-scrolling cursor that allows rows to be updated or retrieved from the active set in consecutive order, or you can DECLARE a scrolling cursor that allows rows to be retrieved in random order. For example, the following DECLARE statement associates a scrolling cursor named **q_curs** with a SELECT statement that retrieves all the rows from the **customer** table:

```
DECLARE q_curs SCROLL CURSOR FOR
  SELECT * FROM customer
```

The following DECLARE statement associates a non-scrolling cursor with a SELECT statement that retrieves customer rows based on a last name that the user supplies:

```
PROMPT "Enter a last name: " FOR last_name

DECLARE cust_curs CURSOR FOR
   SELECT * FROM customer
   WHERE lname MATCHES last_name
```

A cursor name has meaning only from the point at which it is DECLAREd to the end of the source code file. This means that the DECLARE statement for a cursor must physically appear before any statement that makes reference to it.
For example, the following program will not compile because the DECLARE statement for **q_curs** appears after the FOREACH statement that refers to it. (See the following section "Retrieving and Processing Rows" for more information about FOREACH.)

```
DATABASE stores

MAIN

   DEFINE p_customer RECORD LIKE customer.*

   OPEN FORM custform FROM "customer"

   DISPLAY FORM custform

   CALL get_curs()                     # INCORRECT

   FOREACH q_curs INTO p_customer.*

      DISPLAY BY NAME p_customer.*
      . . .

   END FOREACH

END MAIN


FUNCTION get_curs()

   DECLARE q_curs CURSOR FOR
      SELECT * FROM customer

END FUNCTION
```

Retrieving and Processing Rows

Once you have declared a cursor for a SELECT statement, you can use either the FOREACH statement or OPEN, FETCH, and CLOSE to retrieve and process the rows specified by the SELECT statement.

The FOREACH Statement. You can select the rows and execute a series of statements for each row returned by a query using the FOREACH statement. The following example uses a FOREACH statement to retrieve and display rows in the **customer** table:

```
PROMPT "Enter a last name: " FOR last_name

DECLARE q_curs CURSOR FOR
   SELECT * FROM customer
      WHERE lname MATCHES last_name

FOREACH q_curs INTO p_customer.*

   DISPLAY BY NAME p_customer.*
   . . .

END FOREACH
```

When INFORMIX-4GL encounters the FOREACH statement in this example, it runs the query and repeatedly performs the following operations until the active set is exhausted:

- Retrieves the next row from the active set and stores it in the **p_customer** record
- Displays the values in the **p_customer** record on a screen form
- Executes all additional statements within the FOREACH loop

You can use the FOREACH statement when you want to retrieve the rows specified by a SELECT statement and process them in consecutive order. (In such cases, the cursor you specify in the FOREACH statement need not be a scrolling cursor.)

When you need to process the rows returned by a SELECT statement in random order, you must DECLARE a scrolling cursor and use the OPEN, FETCH, and CLOSE statements described in the following section.

OPEN, FETCH, and CLOSE. You can use OPEN, FETCH, and CLOSE when you need to explicitly control the behavior of a cursor:

OPEN puts the cursor in an open state with regard to the SELECT statement. The OPEN statement causes the SELECT statement to be run with the current program variables and leaves the cursor pointing just before the first row of the resulting active set. While the cursor is in an open state, subsequent changes to any program variables that appear in the SELECT statement associated with the cursor do not affect the active set.

FETCH advances the cursor to the specified row (either FIRST, LAST, NEXT, PRIOR or PREVIOUS, ABSOLUTE *n*, or RELATIVE *m*) and retrieves the values from that row. If a FETCH statement moves the cursor before the first row or after the last row, the error variable **status** has the value NOTFOUND (= 100) as does the SQLCODE component of the SQLCA record (see the discussion later in this chapter). NOTFOUND indicates that either end of the active list has been reached.

Note: When you initially fetch a row with a scrolling cursor, all the rows in the active set up to and including the fetched row are placed in a temporary file and remain there until you close the cursor. If you then fetch the same row or any row prior to it, INFORMIX-4GL makes the retrieval from the temporary file instead of from the database. This means that subsequent changes to the database may not be reflected in the active set used by a scrolling cursor.

CLOSE puts the cursor in a closed state and releases the active set. No statements referring to the cursor, other than OPEN, are operative.

The following example shows how to use these cursor management statements to retrieve and display rows in the **customer** table.

```
MAIN

   . . .

   DECLARE q_curs SCROLL CURSOR FOR
      SELECT * FROM customer
         WHERE lname MATCHES last_name

   OPEN q_curs

   FETCH FIRST q_curs INTO p_customer.*

   IF status = NOTFOUND THEN
      CALL mess("No customers found.")
   ELSE
      DISPLAY BY NAME p_customer.*
      CALL viewcust()
   END IF

   CLOSE q_curs

   . . .

END MAIN
```

The main program includes the following statements:

- The DECLARE statement associates a scrolling cursor called **q_curs** with the SELECT statement that retrieves rows from the **customer** table. (The program uses a scroll cursor since the rows specified by the SELECT statement are retrieved in random order.)

- The OPEN statement runs the SELECT statement with the current value of **last_name** and leaves the cursor pointing just before the first row of the active set.

- The FETCH FIRST statement attempts to retrieve the first row of the active set.

- The IF statement displays a message indicating that the active set is empty if the value of the **status** variable is NOTFOUND. Otherwise, it displays the first row on a screen form and calls a function that allows the user to browse through the rows in the active set.

- The CLOSE statement releases the active set after all rows have been processed.

The **viewcust** function displays a menu that lets the user browse through the rows in the active set:

```
FUNCTION viewcust()

   MENU "BROWSE:"

      COMMAND "Next" "View the next customer in the list"
         FETCH NEXT q_curs INTO p_customer.*
         IF status = NOTFOUND THEN
            CALL mess("No more customers in this direction.")
            FETCH LAST q_curs INTO p_customer.*
         END IF
         DISPLAY BY NAME p_customer.*

      COMMAND "Previous" "View the previous customer in the list"
         FETCH PREVIOUS q_curs INTO p_customer.*
         IF status = NOTFOUND THEN
            CALL mess("No more customers in this direction.")
            FETCH FIRST q_curs INTO p_customer.*
         END IF
         DISPLAY BY NAME p_customer.*

      COMMAND "First" "View the first customer in the list"
         FETCH FIRST q_curs INTO p_customer.*
         DISPLAY BY NAME p_customer.*

      COMMAND "Last" "View the last customer in the list"
         FETCH LAST q_curs INTO p_customer.*
         DISPLAY BY NAME p_customer.*

      COMMAND "Exit" "Leave the menu"
         EXIT MENU

   END MENU

END FUNCTION
```

This function consists of a MENU statement that contains the following COMMAND clauses:

Next	includes a FETCH NEXT statement that attempts to retrieve the next row of the active set. The IF statement returns the cursor to the

	last row and displays a message if the value of the **status** variable indicates that the cursor has moved beyond the last row of the active set. The DISPLAY BY NAME statement displays the row retrieved by the appropriate FETCH statement on the screen.
Previous	includes a FETCH PREVIOUS statement that attempts to retrieve the previous row of the active set. The IF statement returns the cursor to the first row and displays a message if the value of the **status** variable indicates that the cursor has moved beyond the first row of the active set. The DISPLAY BY NAME statement displays the row retrieved by the appropriate FETCH statement on the screen.
First	includes a FETCH FIRST statement that retrieves the first row in the active set and displays it on the screen.
Last	includes a FETCH LAST statement that retrieves the last row in the active set and displays it on the screen.
Exit	includes an EXIT MENU statement that terminates the MENU statement.

Note: When you open a cursor that identifies a SELECT statement containing a program variable, INFORMIX-4GL runs the SELECT statement with the current value of the program variable. In the following example, the active set produced by the first OPEN statement differs from the active set produced by the second OPEN statement because the value of **last_name** changes from Baxter to Grant:

```
LET last_name = "Baxter"

DECLARE q_curs SCROLL CURSOR FOR
  SELECT * FROM customer
     WHERE lname MATCHES last_name

OPEN q_curs

. . .

CLOSE q_curs

LET last_name = "Grant"

OPEN q_curs
```

Deleting or Updating the Current Row

You can use special forms of DECLARE, DELETE, and UPDATE to delete or update the current row in an active set. This requires that you use a non-scrolling cursor to process the rows returned by a SELECT statement, and that the SELECT statement not include an ORDER BY clause.

Deleting the Current Row. To delete a row in an active set, you must include a FOR UPDATE clause in the DECLARE statement for a non-scrolling cursor and a WHERE CURRENT OF clause in a subsequent DELETE statement, as follows:

```
PROMPT "Enter a last name: " FOR last_name

DECLARE q_curs CURSOR FOR
   SELECT * FROM customer WHERE lname MATCHES last_name
FOR UPDATE

FOREACH q_curs INTO p_customer.*

   DISPLAY BY NAME p_customer.*

   PROMPT "Do you want to delete this customer (y/n) ? "
      FOR answer

   IF answer = "y" THEN
      DELETE FROM customer WHERE CURRENT OF q_curs
      CLEAR FORM
   END IF

   . . .

END FOREACH
```

The cursor remains between rows after a DELETE WHERE CURRENT OF statement is executed. This means you cannot refer to it in another DELETE or UPDATE statement until you use a FETCH statement to advance the cursor to the next row. (In this example, the FOREACH statement automatically performs the FETCH.)

Updating the Current Row. You can update the current row if you include a FOR UPDATE clause in the DECLARE statement for a non-scrolling cursor and a WHERE CURRENT OF clause in a subsequent UPDATE statement. The following example allows the user to update the address information in the current row:

```
DECLARE q_curs CURSOR FOR
   SELECT * FROM customer
FOR UPDATE

FOREACH q_curs INTO p_customer.*

   DISPLAY BY NAME p_customer.*

   PROMPT "Do you want to change the customer's address (y/n) ? "
      FOR answer

   IF answer = "y" THEN

      INPUT BY NAME p_customer.address1,
                    p_customer.address2,
                    p_customer.city,
                    p_customer.state,
                    p_customer.zipcode
         WITHOUT DEFAULTS

      UPDATE customer
         SET address1 = p_customer.address1,
             address2 = p_customer.address2,
             city = p_customer.city,
             state = p_customer.state,
             zipcode = p_customer.zipcode,
         WHERE CURRENT OF q_curs

   END IF

   . . .

END FOREACH
```

If you specify one or more columns in the FOR UPDATE clause of the DECLARE statement, you can update only those columns in a subsequent UPDATE WHERE CURRENT OF statement. If you do not list columns in a FOR UPDATE OF *column-list* clause, you can update any column retrieved in the select. **INFORMIX-4GL** can usually execute the updates more quickly if you include a FOR UPDATE clause in the DECLARE statement.

The following example allows the user to update the **fname** and **lname** columns of the current row:

```
DECLARE q_curs CURSOR FOR
   SELECT * FROM customer
FOR UPDATE OF fname, lname

FOREACH q_curs INTO p_customer.*

   DISPLAY BY NAME p_customer.*

   PROMPT "Do you want to change ",
      "the contact's name (y/n) ? "
      FOR answer

   IF answer = "y" THEN

      INPUT BY NAME p_customer.fname, p_customer.lnan
         WITHOUT DEFAULTS

      UPDATE customer
         SET fname = p_customer.fname,
             lname = p_customer.lname
         WHERE CURRENT OF q_curs

   END IF

   . . .

END FOREACH
```

The position of the cursor does not change after an UPDATE WHERE CURRENT OF statement is executed.

INSERT Cursors

You can associate a cursor with an INSERT statement as well as a SELECT statement. The INSERT cursor permits more efficient insertion of data into a database by buffering the data in memory and writing to the disk only when the buffer is full. The following statements allow you to declare and manipulate an INSERT cursor. (For complete information about the syntax of each statement, see Chapter 7 of the *INFORMIX-4GL Reference Manual*).

DECLARE	associates a cursor with an INSERT statement. (The INSERT statement may not contain an embedded SELECT statement.)
OPEN	sets up an insert buffer for an INSERT cursor.
PUT	stores a row in the INSERT buffer for later insertion into the database. When you fill the buffer (by executing a series of PUT statements), **RDSQL** automatically inserts the rows into the appropriate table as a block.
FLUSH	forces **RDSQL** to insert the buffered rows into the database without closing the INSERT cursor. You can force the insertion using the FLUSH statement, but you cannot delay insertion by not using the FLUSH statement.
CLOSE	flushes the insert buffer and closes the INSERT cursor.

For example, you can use these cursor management statements to insert customers into the **customer** table block by block, as follows:

```
DECLARE ins_curs CURSOR FOR
   INSERT INTO customer VALUES (p_customer.*)

OPEN ins_curs

LET answer = "y"

WHILE answer = "y"

   INPUT BY NAME p_customer.fname
        THRU p_customer.phone

   LET p_customer.customer_num = 0

   PUT ins_curs

   PROMPT "Do you want to enter ",
      "another customer (y/n) ?  "
      FOR answer

END WHILE

CLOSE ins_curs
```

This example includes the following statements:

- The DECLARE statement associates a cursor called **ins__curs** with an INSERT statement that inserts a row into the **customer** table.
- The OPEN statement sets up the insert buffer for the INSERT cursor.
- The WHILE loop includes statements that insert information entered on a screen form into the **customer** table block by block. Specifically, the INPUT statement allows the user to enter customer information on a screen form and stores it in the **p__customer** record. The PUT statement stores the current values in the **p__customer** record in the insert buffer. If the insert buffer becomes full as the result of a

PUT statement, **RDSQL** automatically inserts the rows into the **customer** table as a block.

- The CLOSE statement inserts any rows that remain in the insert buffer into the **customer** table and closes the INSERT cursor.

When you use an insert cursor, you should CLOSE the cursor to insert any buffered rows into the database before allowing your program to end. The user may lose data if the cursor is not closed properly. For example, if the user presses the INTERRUPT key during input in the following program, **INFORMIX-4GL** closes the INSERT cursor before leaving the program. (This ensures that any remaining rows in the insert buffer are inserted into the database before the program stops.)

```
DEFER INTERRUPT

. . .

DECLARE ins_curs CURSOR FOR
    INSERT INTO customer VALUES (p_customer.*)

OPEN ins_curs

LET answer = "y"

WHILE answer = "y"

    INPUT BY NAME p_customer.fname THRU p_customer.phone
       ON KEY (INTERRUPT)
          CLOSE ins_curs
          EXIT PROGRAM
    END INPUT

    LET p_customer.customer_num = 0

    PUT ins_curs

    PROMPT "Do you want to enter another customer (y/n) ?  "
       FOR answer

END WHILE

CLOSE ins_curs

. . .
```

You can determine whether **INFORMIX-4GL** successfully executed a PUT, FLUSH, or CLOSE statement by examining the values of the **status** and SQLCA.SQLERRD[3] variables. (See the section "SQLCA Record" later in this chapter for more information on these variables.) If **RDSQL** simply puts a row in the insert buffer, it assigns the following values to these global variables:

```
           status =  0
 SQLCA.SQLERRD[3] =  0
```

If **RDSQL** successfully inserts a block of rows into the database as the result of a PUT, FLUSH, or CLOSE statement, it assigns the following values:

```
           status =  0
 SQLCA.SQLERRD[3] =  the number of rows inserted
```

If, as the result of a PUT, FLUSH, or CLOSE statement, **RDSQL** is unsuccessful in its attempt to insert an entire block of rows into the database, it assigns the following values:

```
           status =  a negative number
                     corresponding to the error message
 SQLCA.SQLERRD[3] =  the number of rows successfully inserted
```

Note: If your database has transactions, you must include all PUT statements within a BEGIN WORK—COMMIT WORK or BEGIN WORK—ROLLBACK WORK block. Both COMMIT WORK and ROLLBACK WORK close all open cursors. You should also close an INSERT cursor before committing work so you can check that the insertion was successful.

Dynamic Management

The preceding discussion assumes that you know what the RDSQL statements are when you write your INFORMIX-4GL programs. That is the case for most applications where you are performing predetermined activities on your database. There are several advanced applications, however, where you will not know the statement at compile time. These include the following programs:

- Interactive programs where the user supplies input at runtime from the keyboard
- Programs intended to work with different databases whose structure can vary

In situations like these, you must work with dynamically defined statements. A brief description of the dynamic management statements follows:

PREPARE	takes a character string, interprets it as an RDSQL statement, and assigns it to a statement identifier. Subsequent dynamic management statements refer to the RDSQL statement through the statement identifier.
EXECUTE	runs the previously PREPAREd statement associated with the statement identifier. Use EXECUTE for all prepared statements except the following statements: ● SELECT statements ● INSERT statements that use an insert cursor
DECLARE	has a version that declares a cursor for a PREPAREd SELECT or INSERT statement.

Preparing Statements

You can use the PREPARE statement with either a character string or a character variable that evaluates to an RDSQL statement. The form of the PREPARE statement you choose depends on the type of input (if any) the statement requires. You can use either form of the PREPARE statement if the statement requires no input or input for values. If the statement requires input for RDSQL identifiers such as column names, you must use the PREPARE statement with a character variable.

In general, you can improve the performance of your programs by preparing statements you plan to execute many times. Specifically, you might want to prepare a statement that requires different input each time it is executed.

Note: You can prepare any RDSQL statements except the following statements:

CLOSE	OPEN
DECLARE	PREPARE
EXECUTE	SELECT with an INTO clause
FETCH	

See the section "RDSQL Statement Summary" earlier in this chapter for more information about RDSQL statements. Chapter 1 describes the INFORMIX-4GL statements you cannot prepare.

Statements That Require No Input

If a statement requires no input, you can prepare it from either a character string or character variable. For example, the following statement

```
PREPARE s1 FROM "SELECT * FROM customer"
```

produces the same result as

```
DEFINE sel_stmt CHAR(25)

LET sel_stmt = "SELECT * FROM customer"

PREPARE s1 FROM sel_stmt
```

Statements That Require Input for Values

Similarly, you can use either form of PREPARE when a prepared statement requires input for one or more values.

Preparing a Character String. If you use PREPARE with a character string, you must use a question mark (?) instead of a program variable in the character string as a placeholder for a value. Specifically, the question mark can represent a value or expression in a character string but not an **RDSQL** identifier (such as a column name or table name). Usually, you use a question mark to represent a value in the following clauses:

- The WHERE clause of a SELECT, UPDATE, or DELETE statement:

```
PREPARE sel1 FROM
   "SELECT * FROM customer WHERE lname MATCHES ?"
```

- The VALUES clause of an INSERT statement:

```
PREPARE ins1 FROM
  "INSERT INTO manufact VALUES (?, ?)"
```

- The SET clause of an UPDATE statement:

```
PREPARE upd1 FROM
  "UPDATE customer SET zipcode = ? WHERE CURRENT OF q_curs"
```

When you prepare a statement from a character string, you do not need to supply values for the question marks until you execute the PREPAREd statement (see the section "Executing Prepared Statements" later in this chapter for more information).

Preparing a Character Variable. Alternatively, you can prepare a statement that requires input for values from a character variable. First, you use a LET statement to concatenate the variable(s) containing the input to one or more strings that represent the rest of the statement. Second, you PREPARE the character variable that contains the resulting RDSQL statement.

The following example shows how to use this approach to prepare a statement that selects rows from the **customer** table based on a customer number that the user supplies:

```
DEFINE cust_num INTEGER,
       sel_stmt CHAR(100)

PROMPT "Enter a customer number: "
   FOR cust_num

LET sel_stmt =
   "SELECT * FROM customer WHERE customer_num = ",
      cust_num USING "###"

PREPARE sel1 FROM sel_stmt
```

When INFORMIX-4GL encounters the LET statement in this example, it concatenates a character string containing part of the SELECT statement to the variable **cust_num**, which contains a customer number that the user supplies. INFORMIX-4GL then assigns the resulting string to the large character variable **sel_stmt** and PREPAREs it.

When you use this approach, you must supply input values when you assign the RDSQL statement to the character variable that you will later PREPARE.

Note: If you use the LET statement to concatenate strings to variables that contain CHAR or DATE values, make sure that quotes appear around those values in the resulting character string. To embed a quote in a character string, you must enter a backslash (\) followed by a double quote ("). For example, the LET statement

```
LET sel_stmt = "SELECT * FROM customer WHERE lname MATCHES \"",
    last_name CLIPPED, "\""

PREPARE s1 FROM sel_stmt
```

produces the following character string if the current value of **last_name** is Baxter:

```
SELECT * FROM customer WHERE lname MATCHES "Baxter"
```

In contrast, the statements like the following do not require quotation marks around placeholders for values.

```
DECLARE q_curs CURSOR FOR
    SELECT * FROM customer WHERE lname MATCHES last_name

     PREPARE s1 FROM
          "SELECT * FROM customer WHERE lname MATCHES ?"
```

Statements That Require Input for RDSQL Identifiers

You must use PREPARE with a character variable to prepare a statement that requires data for an RDSQL identifier such as a column name, table name, user name, view name, or synonym. The approach you use is identical to that described in the previous section. First, you concatenate the variable(s) representing the RDSQL identifier(s) to one or more character strings that contain the rest of the statement. Second, you assign the resulting string to a large character variable and PREPARE it.

The following example shows how to use this approach to prepare a statement that grants the CONNECT privilege to a specified user:

```
DEFINE p_user CHAR(12),
       grant_stmt CHAR(50)

PROMPT "Enter the name of user ",
   "to receive CONNECT permission: "
   FOR p_user

LET grant_stmt = "GRANT CONNECT TO ",
   p_user CLIPPED

PREPARE s1 FROM grant_stmt
```

When INFORMIX-4GL encounters the LET statement in this example, it concatenates a character string containing part of the GRANT statement to the character variable **p_user**, which contains a login name. INFORMIX-4GL then assigns the resulting string to **grant_stmt** and PREPAREs it.

Executing Prepared Statements

The method for executing a prepared statement depends upon the kind of statement you want to run. The EXECUTE statement runs any prepared statements except those that follow:

- SELECT statements
- INSERT statements that you want to process as a group

The DECLARE statement has a special form designed to work with prepared SELECT and INSERT statements.

The EXECUTE Statement

If you have prepared a non-SELECT statement from a character variable or from a character string that does not contain question marks, you can run it with a simple EXECUTE statement. Examples follow:

```
PREPARE s1
   FROM "DELETE FROM customer WHERE customer_num = 115"

EXECUTE s1
```

```
LET del_stmt =
   "DELETE FROM customer WHERE customer_num = 115"

PREPARE s1 FROM del_stmt

EXECUTE s1
```

If you prepared a non-SELECT statement from a character string that *does* contain question marks, you must use the EXECUTE statement with a USING clause. This clause consists of the USING keyword followed by one or more program variables representing the values that replace the question marks in the prepared character string.

The EXECUTE statement in the following example executes a DELETE statement using a customer number that the user supplies.

```
PREPARE s1
   FROM "DELETE FROM customer WHERE customer_num = ?"

PROMPT "Do you want to delete a customer (y/n) :  "
   FOR answer

WHILE answer = "y"

   PROMPT "Enter a customer number : "
      FOR cust_num

   EXECUTE s1 USING cust_num

   IF status = 0 THEN
      DISPLAY "Row deleted."
   ELSE
      CALL mess("Unable to delete the customer row.")
   END IF

   PROMPT "Do you want to delete another customer (y/n): "
      FOR answer

END WHILE
```

When INFORMIX-4GL executes the prepared DELETE statement in this example, it substitutes the current value of **cust_num** for the question mark in the character string.

Running Prepared SELECT Statements

If you prepared a SELECT statement from a character variable or from a character string that does not contain question marks, you can use a DECLARE statement with either FOREACH or OPEN, FETCH, and CLOSE. Two examples follow:

```
LET sel_stmt =
   "SELECT * FROM customer WHERE lname MATCHES \"",
   last_name CLIPPED, "\""

PREPARE sel1 FROM sel_stmt

DECLARE q_curs CURSOR FOR sel1

FOREACH q_curs INTO p_customer.*

   DISPLAY BY NAME p_customer.*

   . . .

END FOREACH
```

```
PREPARE sel1 FROM
   "SELECT * FROM customer"

DECLARE q_curs CURSOR FOR sel1

FOREACH q_curs INTO p_customer.*

   DISPLAY BY NAME p_customer.*

   . . .

END FOREACH
```

If you prepared a SELECT statement from a character string that does contain one or more question marks, you must use the DECLARE statement with an OPEN statement that includes a USING clause. As described previously, this clause consists of the USING keyword followed by one or more program variables representing the values that replace the question marks in the character string. An example follows:

```
PREPARE sel1 FROM
   "SELECT * FROM customer WHERE zipcode MATCHES ?"

DECLARE q_curs CURSOR FOR sel1

PROMPT "Enter a zipcode: " FOR zip

OPEN q_curs USING zip

WHILE TRUE

   FETCH q_curs INTO p_customer.*

   IF status = NOTFOUND THEN
      EXIT WHILE
   END IF

   DISPLAY BY NAME p_customer.*

   . . .

END WHILE

CLOSE q_curs
```

When INFORMIX-4GL opens the cursor for the prepared SELECT statement in this example, it substitutes the current value of **zip** for the question mark in the character string.

A Note on Preparing and Executing SELECT Statements for Update:

A previous section entitled "Deleting or Updating the Current Row" describes how to use the DECLARE FOR UPDATE statement and a subsequent DELETE or UPDATE statement to delete or update the current row. If you want to use DECLARE FOR UPDATE with a *prepared* SELECT statement, make sure the FOR UPDATE clause appears as part of the prepared character string or character variable and not as part of the DECLARE statement. An example follows:

```
PREPARE s1 FROM "SELECT * FROM customer FOR UPDATE"

DECLARE q_curs CURSOR FOR s1

FOREACH q_curs INTO p_customer.*

    . . .

DELETE FROM customer WHERE CURRENT OF q_curs

END FOREACH
```

Running Prepared INSERT Statements

If you have PREPAREd an INSERT statement, you can run it by using the EXECUTE statement or the PUT statement. A previous section, "The EXECUTE Statement," describes how to use the EXECUTE statement if you want **RDSQL** to insert one row into the database at a time. This section explains how to DECLARE a cursor and use the PUT statement to insert rows into the database through an insert buffer.

If you prepared an INSERT statement from a character variable or from a character string that does not contain question marks, you can use the DECLARE, OPEN, FLUSH and/or CLOSE statements with a simple PUT statement, as follows:

```
PREPARE s1 FROM
   "INSERT INTO manufact VALUES ("WLS", "Willis")

DECLARE icurs CURSOR FOR s1

OPEN icurs

PUT icurs

. . .

CLOSE icurs
```

If you prepared an INSERT statement from a character string that does contain one or more question marks, you must use the DECLARE, OPEN, FLUSH, and/or CLOSE statements with a PUT statement that includes a FROM clause. This clause consists of the FROM keyword followed by one or more program variables representing the values that replace the question marks in the character string. An example follows:

```
PREPARE s1 FROM
   "INSERT INTO customer (customer_num, company) VALUES (0, ?)"

DECLARE ins_curs CURSOR FOR s1

OPEN ins_curs

LET answer = "y"

WHILE answer = "y"

   Prompt "Enter a customer: " FOR p_customer.company

   PUT ins_curs FROM p_customer.company

   PROMPT "Do you want to enter another customer (y/n) ?  "
      FOR answer

END WHILE

CLOSE ins_curs
```

When INFORMIX-4GL executes the PUT statement in this example, it substitutes the current value of **p_customer.company** for the question mark in the prepared INSERT statement and stores the row in the insert buffer for later insertion into the database.

Data Access

A user has access to the database, a table, and to specific columns within a table only when the Database Administrator (DBA) or the owner of the table specifically grants these privileges. You can temporarily limit access to a table by executing the LOCK TABLE statement. (Under transactions, RDSQL locks the affected rows until the transaction is complete. Explicit table/record locking is generally not required.) The following RDSQL statements affect data access

GRANT	grants database access privileges to specific users or to the public.
REVOKE	removes database access privileges from specific users or from the public.
LOCK TABLE	limits access to the table to the current user only or allows other users only to read the table. Use the LOCK TABLE statement only when making major changes to a table in a multi-user environment and when simultaneous interaction with the table by another user would interfere. LOCK TABLE decreases the accessibility of the database, since it prevents other users from accessing the table. If the database has transactions, you must issue a BEGIN WORK statement before you can issue the LOCK TABLE statement.
UNLOCK TABLE	restores access to a previously LOCKed table.

See the later section "User Status and Privileges" for further information.

Data Integrity

RDSQL provides data integrity in several ways. You can prevent users of the database from changing the same row at the same time. The UPDATE statement automatically does this. In addition, you can explicitly locking a table, or access the row from within a transaction. You can also guarantee data integrity by using the recovery feature, which is implemented with transactions, or by using audit trails. This section describes transactions and audit trails; RDSQL locking statements are considered in the section "Locking" later in this chapter.

Transactions

RDSQL provides data integrity on the database level by implementing transactions. A transaction is a series of operations on a database (RDSQL statements) that you want to be completed entirely or not at all. Examples of transactions are abundant in bookkeeping where several operations on several different accounts must be made as a unit or the books will be out of balance. Your INFORMIX-4GL programs can ensure the data integrity of your database by using the following statements:

BEGIN WORK	marks the beginning of a transaction.
COMMIT WORK	marks the end of a transaction by authorizing all changes to the database since the last BEGIN WORK statement. COMMIT WORK releases all row and table locks.
ROLLBACK WORK	marks the end of a transaction by revoking all changes to the database since the last BEGIN WORK statement.
START DATABASE	initiates a new transaction log file.

ROLLFORWARD DATABASE	uses a transaction log file to restore a database from backup.

To use transactions, your database must have a *transaction log.* You can create the transaction log by including the WITH LOG IN clause in the CREATE DATABASE statement. This statement creates not only the database but also a transaction log file that keeps track of all modifications to the database. Alternately, you can execute the START DATABASE statement and initialize a new database transaction log.

When you want to perform a series of operations that you consider a unit, you issue the BEGIN WORK statement. This statement causes all subsequently altered rows of the database tables to be locked against modification by others (although others can view them). When you are satisfied that the series of operations has produced the desired results, you terminate the transaction with a COMMIT WORK statement. If you are not satisfied with the results, you can terminate the transaction with a ROLLBACK WORK statement. The ROLLBACK WORK statement restores the database to the state that existed when you issued the BEGIN WORK statement—with an important exception. All data definition statements are treated as singleton transactions; that is, if they were executed successfully, they are committed and are not rolled back. (RDSQL data definition statements are those that alter the number or names of tables or views, the number or names of columns, the data types, or the indexes. See the section "Data Definition" earlier in this chapter for more information about these statements.) It is recommended that all data definition statements not appear within a transaction.

Both the COMMIT WORK and the ROLLBACK WORK statements unlock the rows and make them accessible for modification by others.

The number of rows that can be locked at one time by all users is limited. The actual limit depends on your operating system. Try to restrict the definition of a transaction to a few statements that involve only a few rows. If you expect that the number of rows entering into the transaction will be large, LOCK the tables involved until the transaction is completed.

If the database has transactions, you must issue a BEGIN WORK statement before you can issue the LOCK TABLE statement.

Note: The BEGIN WORK, COMMIT WORK, and ROLLBACK WORK statements only work when you CREATEd the database with the WITH LOG IN option or STARTed the database with a transaction log file. A database created without a transaction log is described as a database "without transactions." A database created with a transaction log is described as a database "with transactions."

If you open or create a database with transactions, but do not execute the BEGIN WORK statement, **RDSQL** treats each statement as a singleton transaction. Each statement, if it executes successfully, is committed, and the database is permanently altered. If the statement fails, there is an automatic rollback to the status before the statement.

You must execute cursor manipulation statements inside a transaction if your database was created or started with transactions. You should execute the BEGIN WORK statement before opening a cursor. The COMMIT WORK and ROLLBACK WORK statements close all open cursors, although it is not recommended that you use them for this purpose.

You cannot ROLLBACK any of the data definition statements nor GRANT or REVOKE statements. For this reason you should execute them as singleton transactions.

If you create a database without transactions, there is no automatic recovery from a situation where you want to treat several database operations as a single unit of work. For example, under transactions, you can UPDATE several rows as a single unit of work. If the UPDATE fails after changing some rows but not all, you can ROLLBACK the transaction to the original state where no rows are modified. Without transactions, you must take explicit action to restore the updated rows.

Transaction Log File Maintenance

The transaction log file can become quite large and, periodically, the Database Administrator (DBA) will want to archive it on tape and initiate another log file. At the same time, the DBA should also create a backup of your database. Generally speaking, every log file must have a corresponding archive copy of the database. After backing up the log file and the database, the DBA must specify an empty log file. To reuse the same log file, the DBA can create an empty log file with the same name as the old one. In UNIX, you can do this with the following command:

```
cat /dev/null > logfile
```

To change the name of the log file, the DBA must execute the START DATABASE statement just before making a backup of the database. The START DATABASE statement locks the database in EXCLUSIVE MODE while it is operating so that no further changes can be made. If START DATABASE fails, no database is open.

If the database is without transactions and you want to use transactions, the DBA must execute the START DATABASE command just before making a copy of the database.

If there is a backup copy of the database and a transaction log file that begins with the operations executed immediately after the backup was made, the DBA can bring the backup database up to date with the ROLLFORWARD DATABASE statement. This statement recovers the database through the last terminated transaction. The DBA must load the backup database files and execute the ROLLFORWARD DATABASE statement. After rolling the database forward, the DBA is the only one who has access to the database, since it is left in an exclusive mode. This state allows the DBA to check the database for errors before making it generally available. Logging does not occur during this checking phase. The DBA must close the database when it has been restored correctly.

Audit Trails

An audit trail is a file that contains a history of all additions, deletions, updates, and manipulations to a database table. An audit trail serves a purpose similar to that of a transaction log: each is used to maintain a record of modifications to a database, and each can be used to update backup copies of a database.

Three audit trail statements are available to protect the integrity of a table:

CREATE AUDIT	creates an audit trail for a table.
DROP AUDIT	removes the audit trail on a table.
RECOVER TABLE	restores a table using the audit trail.

Creating an Audit Trail

Use the CREATE AUDIT statement to create an audit trail file and to begin writing the audit trail. The format is

CREATE AUDIT FOR *table-name* IN "*pathname*"

where *table-name* is the name of the table for which you want to create an audit trail file and *pathname* is the full pathname of the audit trail file. The audit trail file should be on a physical device other than the one that holds the data so that a system failure affecting the device that holds the data does not also damage the audit trail. If your computer system has more than one hard disk, the audit trails should be written to a disk not containing the data.

To use the audit trail, make a backup copy of the table *after* you have executed a CREATE AUDIT statement but *before* you have made any changes to the table. Once you have started the audit trail and have made a backup, you are ready to work with the table.

You can drop and create an audit trail file whenever you want. Drop and create the audit trail files just before you make a complete backup of the device containing the data file. If a system failure should occur, you can use the audit trail to backup the table from the time of the last backup to the time the failure occurred.

Recovering a Table

In the event of a system failure, you can use the RECOVER TABLE statement to restore a database table by using a backup copy of the table and an audit trail file. You must first restore a backup copy of the table. The backup copy must be in the original state that it was in when the audit trail was started. If it is not in the original state, the recovery fails. The format of the recovery statement is

```
RECOVER TABLE table-name
```

where *table-name* is the name of the table you want to recover.

Once you recover the table, use the DROP AUDIT statement to remove the contents of the audit trail file. Then run the CREATE AUDIT statement to start a new audit trail file. Finally, make a new backup copy of the table.

Comparison of Transactions and Audit Trails

Transactions provide data integrity in two ways. First, they guarantee that RDSQL statements are either successfully completed or are completely canceled. If, for example, you update several rows of one or more tables within a transaction, the entire update is guaranteed either to succeed by updating all rows, or to fail without changing any rows. Second, the transaction log can be used to recover an entire database.

Audit trails are associated with individual tables. They do not guarantee that modifications to several rows of a table either succeed entirely or fail without any effect. An audit trail file can be used only to recover the table for which it is created.

You should consider using audit trails in place of a transaction log only when you have one or a few critical tables and you do not need the additional facilities provided by transactions. If you need to maintain the integrity of the database as a whole, or need the guarantee that RDSQL statements are executed as a unit either entirely or not at all, you must use transactions.

User Status and Privileges

When you create a database, you are automatically the Database Administrator (DBA) of that database and are the only one who has access to the database. Another user does not have access to a database until you grant the CONNECT privilege to that person. Another user cannot create or drop tables and indexes unless granted the RESOURCE privilege. Only the Database Administrator (you, initially) can grant these privileges. You can also grant the DBA privilege to another user. The DBA privilege extends all the powers of the Database Administrator to the grantee, including the ability to alter the system tables; to drop, start, and roll forward the database; and to grant CONNECT, RESOURCE, and DBA privileges to others.

If you have the RESOURCE privilege, you have the CONNECT privilege by default. With the DBA privilege, you have both the RESOURCE and CONNECT privileges. You can only revoke the privilege of a DBA grantee; you cannot revoke your own DBA privilege. If you, as the creator of a database, grant DBA privileges to another user, that user can revoke the DBA privilege from you, the database creator. This last property permits the transfer of authority from the maker of the database application to the person who has responsibility for maintaining the database.

RDSQL allows the CONNECT and RESOURCE privileges to be granted to the PUBLIC in addition to specifically named users.

In addition to these database-level privileges, there are a collection of table-level privileges that can be granted by the owner of a table. These permit the grantee access to specific columns for the purpose of executing SELECT or UPDATE statements, or give the grantee authority to insert new rows, delete old rows, create indexes, and alter the structure of the table.

Several of the RDSQL statements (ALTER TABLE, ALTER INDEX, DROP INDEX, DROP TABLE, DROP VIEW, GRANT, RENAME COLUMN, RENAME TABLE, REVOKE) can be executed only by the DBA or by the owner of the table or index specified in the statement. (You can give others the privilege of executing the ALTER TABLE, GRANT, and REVOKE statements with certain restrictions.) The owner of a table is the person (login name) who executed the CREATE TABLE statement. The owner of an index is the one who executed the CREATE INDEX statement. Execution occurs when the compiled INFORMIX-4GL program containing the CREATE statements is run, not when the INFORMIX-4GL program is compiled.

Indexing Strategy

There are two major purposes for creating an index on columns of a database table: to speed sorting of rows and to optimize the performance of queries. When your application writes reports involving complex queries through a large database, significant time savings can result from judicious indexing. The drawback to having an index is that indexes slow down the process of inserting new data into the database. When you update a table, its indexes may also be modified. This is not a problem when you are adding information interactively, a row at a time, but can become serious when it is necessary to insert a large number of rows from one table into another.

The solution to this potential conflict between needs is to take a dynamic approach to indexing. One of the advantages of a relational database built upon **C-ISAM** is that you do not have to decide issues like which columns to index at the time that you create your tables. You should write your applications to create indexes when you need them and to drop them when they get in the way. It takes time to create an index on a table already containing data, and you should create only those indexes that optimize the queries you make. For example, by judicious scheduling, you can create your indexes in anticipation of batch report writing during the night and drop them the next morning before there are huge data-entry needs.

The following are hints for strategic indexing. Although the last two refer to a single query, they apply when you anticipate making a number of queries with the same qualities.

- Do not create indexes for small tables with fewer than 200 rows. The speed you gain from using an index does not overcome the time required to open and search the index file on small tables.

- Do not create indexes on a column that has only a few possible values. Such columns are those that contain data like sex, marital status, yes/no responses, or zip codes in a small city. Because data like this produces skewed indexes, indexing can cause the optimizing strategy of **RDSQL** to fail and queries to take longer than if the columns were not indexed. If you have a frequent need to have data sorted on columns with a small range of possible values, create a temporary table of the sorted data. Another approach is to redesign the database with separate tables for each alternative value.

- If the WHERE clause of a SELECT statement imposes a condition on a single column, put an index on that column. If there are conditions placed on several columns, make a composite index on all the affected columns. For the SELECT statement

  ```
  SELECT * FROM items WHERE order_num > 1015
  ```

  put an index on **order_num**. For the statement

  ```
  SELECT * FROM items
     WHERE order_num = 1015
        AND total_price > 1000.00
  ```

  create a composite index on both **order_num** and **total_price**.

- If the WHERE clause of a SELECT statement has a join condition between a single column in one table and a single column in another table, create an index on the column in the table with the larger number of rows. If several columns of one table have join conditions with several columns in another table, create a composite index on the affected columns of the table with the larger number of rows. For the SELECT statement

  ```
  SELECT * FROM items, stock
     WHERE items.stock_num = stock.stock_num
  ```

  place an index on **stock_num** in the **items** table, since it has many more rows than the **stock** table. You should execute the UPDATE STATISTICS statement before the SELECT statement so that RDSQL knows the current size of the tables.

For the statement

```
SELECT * FROM items, stock
   WHERE items.stock_num = stock.stock_num
     AND items.manu_code = stock.manu_code
```

put a composite index on **stock_num** and **manu_code** the **items** table.

Auto-Indexing

If you execute a SELECT statement that includes a join between two tables and there are no indexes on the joined columns, RDSQL creates a temporary index on the table with the larger number of rows before performing the join. The index disappears when the query finishes. This enhancement is transparent to the user except for a dramatic improvement in the speed of unindexed joins.

Clustered Indexes

Since both UNIX systems and DOS systems extract information from the disk in blocks, the more rows that are physically on the same block and are already in the order of an index, the faster an indexed retrieval proceeds. Ordinarily, no relationship need exist between the physical order of the data in the **.dat** file and the order in an index. You can, at least temporarily, cause the physical order in the table to be the same as the ordering in an index through *clustering*.

RDSQL orders, or clusters, the physical data in a table when you create a new index by executing a variant of the CREATE INDEX statement or when you execute the new ALTER INDEX statement for an existing index. Since users who have access to the table can add additional rows or update the information in existing rows, a table that you cluster according to an index does not stay that way. Over time, you can expect the benefit of an earlier clustering to disappear and you may want to cluster the table again using an ALTER INDEX TO CLUSTER statement.

Since a table can have only one physical order, you can have only one clustered index on a table at any given time. You can change the physical order to reflect a different index by executing two ALTER INDEX statements:

1. Execute an ALTER INDEX TO NOT CLUSTER statement to release the cluster attribute from the first index.

2. Execute an ALTER INDEX TO CLUSTER statement to attach the cluster attribute to the second index.

You cannot execute the ALTER INDEX or CREATE INDEX statements on a view.

Locking

INFORMIX-4GL uses locking to prevent different users from executing conflicting operations on the same data. Without locking, for example, two users may be allowed to update the same row at the same time. In this situation, the computer memory contains two different versions of the rows (the one updated by user A and the one updated by user B). Without some method of *concurrency control*, the user whose row is the last one actually written to the file "wins" and overwrites the other user's changes.

The following **RDSQL** statements are used to control locking:

LOCK TABLE	limits access to the table to the current user only or allows other users only to read the table. Use the LOCK TABLE statement only when making major changes to a table in a multi-user environment and when simultaneous interaction with the table by another user would interfere. LOCK TABLE decreases the accessibility of the database, since it prevents other users from accessing the table. If the database has transactions, you must issue a BEGIN WORK statement before you can issue the LOCK TABLE statement.

UNLOCK TABLE	restores access to a previously LOCKed table.
SET LOCK MODE	alters the locking strategy either to fail when a row is already locked or to wait for the lock to be released before proceeding.

In addition, in the rare instance where you need to limit access to the entire database to a single user, you can open the database in EXCLUSIVE mode.

INFORMIX-4GL provides two levels of locking:

- Row-level or record-level locking
- Table-level or file-level locking

RDSQL performs row-level locking implicitly. The locking strategy may differ slightly, depending upon whether or not the database uses transaction management. Data definition statements, such as ALTER TABLE, CREATE INDEX, and so on, use implied table-level locking. You can explicitly specify table-level locking. The following sections describe each level of locking and the methods for its use.

Row-Level Locking

Ordinarily, **RDSQL** locks a row when you execute an UPDATE statement, or when you execute a FETCH statement and the cursor is DECLAREd with a FOR UPDATE clause. If the UPDATE statement affects only one row, **RDSQL** releases the lock immediately after performing the update. This prevents two programs from attempting to update the same record at the same time. One program receives the lock and can proceed with the update. The other program either fails in its attempt or waits for that program to release the lock. (See the section "Wait for Locked Row" later in this chapter.)

If the UPDATE statement affects more than one row, **RDSQL** uses the same row-locking strategy. As soon as it completes an update, it releases the lock, then locks and updates the next record. When the UPDATE finishes, all records are unlocked.

If you want more control over the update of multiple records, you can DECLARE a cursor FOR UPDATE. The WHERE clause of the SELECT statement specifies the rows you want to update. After you OPEN the cursor and FETCH a record, that record remains locked until you either CLOSE the cursor or FETCH the next record.

Row-Level Locking in Transactions

If your database uses transaction management, rows that you INSERT, UPDATE or DELETE within a transaction remain locked until the end of the transaction. The end of a transaction is either a COMMIT WORK, where all modifications are made to the database, or a ROLLBACK WORK, where none of the modifications are made.

RDSQL locks a row when it is selected for update. For example, if you DECLARE a cursor FOR UPDATE, the FETCH statement locks the row. The row remains locked until the end of the transaction.

Table-Level Locking

Use table-level locking to lock an entire table and prevent others from altering or seeing rows in that table.

You may want to use this form of locking during batch operations that affect every row in a table, for example. If the operations must be completed as a single transaction, it may be more efficient to lock the entire table at the beginning of the transaction. Normally, under transactions, INFORMIX-4GL locks each row as it is UPDATEd, DELETEd, or INSERTed. If you lock the entire table, however, INFORMIX-4GL does not use row-level locking because it is unnecessary. As a result, you are not likely to reach the limits that your operating system may place on the number of rows that can be locked at any one time.

RDSQL performs table-level locking automatically as part of the following statements: ALTER TABLE, DROP TABLE, CREATE INDEX, ALTER INDEX, and DROP INDEX.

The LOCK TABLE statement has two extensions:

- If you lock the table IN SHARE MODE, other users are able to SELECT data from the table, but they are not able to INSERT, DELETE, or UPDATE rows in the table.
- If you lock the table IN EXCLUSIVE MODE, other users are not able to access the table at all until you execute an UNLOCK TABLE statement.

Because locking an entire table prevents others from adding or altering data in the table, use this feature sparingly. Lock the entire table only when row-level locking (as described in the previous section) is not sufficient.

Wait for Locked Row

If another user locks a row in a table at the row level and you attempt to alter or delete that row (or examine it with SELECT statement FOR UPDATE), RDSQL returns an error stating that the row is locked. If you prefer that RDSQL wait on any locked row until the competing process unlocks it, you can execute the SET LOCK MODE TO WAIT statement. From then on, your request waits until RDSQL unlocks the requested row, and you do not receive an error code.

Caution! If for some reason the competing process fails without unlocking the row, your process deadlocks. Consequently, use the SET LOCK MODE statement with caution.

If another user locks a table IN EXCLUSIVE MODE and you attempt to alter, to delete, or even to read a row in the table, RDSQL returns an error code. The wait-for-lock feature applies only on multi-user systems that support record-level locking.

NULL Values

The basic purpose of introducing NULL values in a database is to indicate when no value has been assigned to a particular column in a particular row of a table. Your reasons for not having assigned a value could include not knowing the correct value or that no value yet exists. The NULL may also indicate that no value is appropriate for a given column because of the values that were entered into other columns.

As an example, consider entering data for a bank customer who is requesting a loan. If the customer, Mr. Farthing, is not employed, the **employer** column in the **client** table will have no entry for this customer. This CHAR column will have the value NULL. The **hire_date** column is meaningless if Mr. Farthing is not employed. There is no appropriate date to enter; the value is NULL.

Default Values

In RDSQL, the default value for a column is NULL. RDSQL makes a distinction for numeric values between zero and NULL and for character values between blanks and NULL. You do not need to know how RDSQL implements the value NULL to make use of it.

By definition, type SERIAL columns can never contain the NULL value. Columns of type SERIAL always contain integers greater than or equal to one.

You can insist that a column of any type not have NULL values by using the NOT NULL clause in the CREATE TABLE statement. RDSQL will prevent a NULL from being entered into any column that is declared NOT NULL.

You cannot, however, use a NOT NULL clause in an ALTER TABLE statement when you add a new column. The reason is that RDSQL enters a NULL value into that column for all rows that already exist.

A column for which you create a unique index can have, at most, one NULL value.

Note for Users with an RDSQL Version 1 Database:

When no value is provided for a column entry in a row of a table in an RDSQL Version 1 database, RDSQL enters a blank for type CHAR columns, zeroes for numeric columns, and a very large negative value for type DATE columns. Since zero could well be an acceptable value for a numeric column (for example, the value for a type MONEY column), there is no way to distinguish an unknown value from zero.

To incorporate an existing RDSQL Version 1 database into INFORMIX-4GL programs, you must execute the **dbupdate** utility described in Appendix I. (Appendix I also describes how you can avoid using NULL values.)

The NULL in Expressions

If any value that participates in an arithmetic expression is NULL, the value of the entire expression is NULL. For example, consider the following query:

```
SELECT order_num, ship_charge/ship_weight
FROM orders
WHERE order_num = 1023
```

If **ship__weight** is NULL because the order with number 1023 is new and the shipping charge has not yet been determined, the value returned for **ship__charge/ship__weight** will also be NULL.

The situation is different when you use one of the aggregate functions (see Chapter 7 for a description of the aggregate functions). COUNT(*) counts all rows, even if the value of every column in the row is NULL. COUNT(DISTINCT *column-name*), AVG, SUM, MAX, and MIN ignore rows with NULL values for the column in their argument and return the appropriate value based on the rest of the rows. However, if a column contains only NULL values, then COUNT(DISTINCT *column-name*) returns zero, and the other four aggregate functions return NULL for that column.

The NULL in Boolean Expressions

In order to incorporate NULL values into Boolean expressions, it is necessary to enlarge the number of truth values from simply true and false to include unknown. If one of the expressions of a Boolean expression is NULL, the truth value of the Boolean expression is unknown. For example, the Boolean expression,

```
ship_charge/ship_weight < 5.0
```

has the truth value unknown for the order in the previous example.

If you combine Boolean expressions using the operators AND, OR, and NOT, the following tables give the resulting truth value (where T corresponds to true, F to false, and ? to unknown).

AND	T	F	?
T	T	F	?
F	F	F	F
?	?	F	?

OR	T	F	?
T	T	T	T
F	T	F	?
?	T	?	?

NOT	
T	F
F	T
?	?

The NULL in WHERE Clauses

If the Boolean expression in a WHERE clause evaluates to unknown for a particular row, RDSQL treats the search condition as not satisfied and does not select or modify that row.

Consider this clause

```
WHERE ship_charge/ship_weight < 5
    AND order_num = 1023
```

The row where **order_num** = 1023 is the row where **ship_weight** is NULL. Since **ship_weight** is NULL, **ship_charge/ship_weight** is also NULL, and the truth value of **ship_charge/ship_weight** < 5 is unknown. Since **order_num** = 1023 is true, the preceding AND truth table states that the truth value of the entire search condition is unknown. Consequently, that row will not be chosen. If the search condition had used an OR in place of the AND, the search condition would be true.

You can select (or reject) rows containing NULL values with a new type of search condition:

---

column IS [NOT] NULL

You must use the keyword IS. It is not permitted to write the condition as follows.

---

column = NULL	(Incorrect)
column != NULL	(Incorrect)

If you perform a join between two tables using the WHERE clause,

```
WHERE column1 = column2
```

RDSQL will not select the rows where either **column1** or **column2** is NULL. In particular, no row will be returned if both **column1** and **column2** are NULL. This is merely a special case of the more general rule that Boolean expressions containing NULL values have an unknown truth value.

Similarly, if a subquery returns a single NULL value, the search condition evaluates to unknown.

The NULL in ORDER BY Clauses

For the purpose of sorting rows using the ORDER BY clause, the NULL value is treated as being less than a non-NULL value. When the ordering is ASC, the NULL values come first; when the ordering is DESC, the NULL values come last.

The NULL in GROUP BY Clauses

RDSQL treats each row containing a NULL value in the column being GROUPed BY as in a separate group. This is consistent with the fact that each NULL value is really unknown. There is no reason to treat all NULL values as being the same.

The NULL Keyword in INSERT and UPDATE Statements

When you execute the INSERT statement, RDSQL will insert the NULL value into all columns for which you do not provide a value or for all columns not listed explicitly. Since the *value-list* of the INSERT statement must be the same length as the *column-list*, you can use the keyword NULL to indicate that a column in *column-list* should be assigned a NULL value.

```
INSERT INTO orders (order_num, order_date,
    customer_num)
    VALUES (0, NULL, 123)
```

All other columns in the **orders** table will be filled with NULL values. Similarly, you can use the NULL keyword to modify a column value when using the UPDATE statement. For a customer whose previous address required two address lines, but now requires only one, you would use the following entry:

```
UPDATE customer
    SET address1 = "123 New Street",
        address2 = NULL,
        city     = "Palo Alto",
        zipcode  = "94303"
    WHERE customer_num = 134
```

Views

Views are constructs on a database that allow you to do the following tasks:

- Provide different users with different windows (called "views") on the data in the database. A single view may involve columns from different tables or may show values that are functions of the values from the columns. A view has a name and looks to a user as if it were a table. The user may query a view, for example, using the same syntax as though the view were a table in the database.

- Limit access to sensitive data by allowing users to see only aggregate information. With the GRANT and REVOKE statements, you can prevent a user from seeing any salary data in a personnel table. With a view, you can allow the user to see average salaries in various groups, but still protect the individual salary data.

- Permit users to update, insert, and to delete data in the database as though the data were organized as it appears in a view. You can also examine through a view the changes made in a real table of the database.

Views are therefore dynamic windows into the database and are not static snapshots. They differ in this respect from a temporary table created by the INTO TEMP clause of a SELECT statement or the CREATE TEMP TABLE statement. Such temporary tables show you only the state of the database when the temporary table was created.

Although views appear to be tables in the database, there are several important ways in which they differ. You cannot use a view in place of a table in a form specification file. You cannot create an index on a view. There are conditions under which you cannot update or modify the data perceived through a view. An obvious case occurs when the "column" seen in a view is really an expression generated from actual database tables. Generally speaking, there is no way to determine the appropriate change in the underlying columns involved in such an expression if you want to change the value of the "column."

The next sections describe how to create and delete views, how to query the database through views, how to modify the database through a view, and how to set up permissions for a view.

Creating and Deleting Views

You must use the CREATE VIEW statement to create a view. (See Chapter 7 for complete information about the CREATE VIEW statement.) A view is determined by a SELECT statement that returns the "table" that defines the view. You cannot use the UNION operator (see Chapter 7 for the definition of the UNION operator) in the definition of a view. The SELECT statement is stored in the **sysviews** system catalog. When you subsequently make reference to a view in another statement, RDSQL performs the defining SELECT statement in executing the new statement.

You can use the same column names as in the underlying table for the view or you can assign new names. When a column in a view is the evaluation of an expression or is not unique (because, for example, you have included all the columns of a join, including the columns that define the join), you must supply new names. These column names are stored in **syscolumns** along with the column names of regular tables.

You can delete a view by executing the DROP VIEW statement. When you drop a view, you also drop all views that were defined in terms of that view.

Querying Through Views

You can make queries involving views exactly as though they were tables in the database. If possible, RDSQL first combines the view-defining SELECT statement with the query to create a new SELECT statement and then executes the new statement. Otherwise, it creates the view as a temporary table and applies the query to the table. RDSQL may detect errors during either of these phases.

Modifying Through Views

Like querying through views, you may use the INSERT, UPDATE, and DELETE statements with views. RDSQL combines the view-defining SELECT statement with the view-referring statement and then executes it. The following restrictions apply to modifying tables through a view:

- You cannot modify the database through a view if the view definition involves joins, the GROUP BY clause, the DISTINCT keyword, or an aggregate function. If any of these features is present in the view definition, the creator of the view cannot execute INSERT, DELETE, or UPDATE statements on the view. You may, however, define a view using a subquery that refers to another table and can often circumvent the restriction on joins (see the section "Data Constraints Using Views," later in this chapter).

- A view column may be UPDATEd only if it is derived directly from a table of the database and not as a result of an expression. Expression-derived columns are called "virtual" columns. You cannot INSERT rows through a view that contains virtual columns, although you may DELETE a row that contains a virtual column.

- You may not execute the ALTER TABLE, CREATE INDEX, ALTER INDEX, or UPDATE STATISTICS statements on a view. You do receive the benefit of existing indexes on the underlying tables.

You may use an INSERT statement on a view that shows only a portion of an underlying table. When you do so, the unmentioned columns of the underlying table will receive NULL values. If one of the unmentioned columns does not permit NULL values, RDSQL will not permit you to INSERT to the view.

If you drop a column of an underlying table of a view that you have defined in terms of that column, RDSQL issues an error if you subsequently make reference (other than DROP VIEW) to the view.

Unless you create the view with a WITH CHECK OPTION clause, it is possible to INSERT or UPDATE data into a database through a view that does not satisfy the limitations on the view. A row INSERTed or UPDATEd in this manner is no longer accessible through the view. For example, a view could be created that allows the user access only to customers from Palo Alto. If, when using the view, the user creates a new row with a customer from Menlo Park, the user cannot select the row through the view. If the **city** column on an existing row is UPDATEd to Menlo Park, the row disappears from the view. The WITH CHECK OPTION clause on the CREATE VIEW statement causes RDSQL to reject an UPDATE or INSERT that violates the restrictions of the view.

You must be careful when you UPDATE a table through a view that may contain duplicate rows. Duplicate rows can occur in a view even if the underlying table has unique rows. If a view is defined on the **items** table and contains only the columns **order_num** and **total_price**, the view contains duplicate rows if two items from the same order have the same total price. If you put the cursor on one of the rows where **total_price** = $1234.56 and update the **total_price** to $1250.00 through the view, you have no way of knowing which item you have increased.

Privileges with Views

When you create a view, you receive the same privileges that you had on the underlying tables. If you have these privileges with the GRANT OPTION (see Chapter 7), you may grant privileges on your view to other users.

If the view is built on more than one table, you can have only the SELECT privilege since multi-table views do not permit you to INSERT, DELETE, or UPDATE. You must have the SELECT privilege on all of the columns from which a multi-table view is derived to have the SELECT privilege on the entire view. If, as a result of these restrictions, you have no privileges on a view, the CREATE VIEW statement returns an error code.

Data Constraints Using Views

The purpose of data constraints is to ensure that all data entered into the database satisfies pre-assigned limitations. Through a form in INFORMIX-4GL, data entry can be controlled with the INCLUDE attribute that lists values and ranges of values permitted for a column. The values entered into the **syscolval** table in the **include** column serve a similar purpose (see Chapter 3). In both of these cases, however, the list of allowed values is static and is dependent only on the designated column.

It is often desirable to define allowed value ranges dynamically, based on the values in other columns or even in other tables. The existence of views and, specifically, the WITH CHECK OPTION clause permits the DBA to control the entry of data into the database. This is most easily demonstrated with an example taken from the **stores** database.

Suppose you want to ensure that no item

- Has a value of more than $20,000
- Is for stock that does not exist

The first step is to create the following view:

```
CREATE VIEW safe_items AS
   SELECT * FROM items
      WHERE total_price < 20000 AND
         EXISTS (SELECT stock_num, manu_code
                    FROM stock
                    WHERE stock.stock_num
                       = items.stock_num
                      AND stock.manu_code
                       = items.manu_code)
   WITH CHECK OPTION
```

If you do all data entry and data modification through the **safe_items** view, RDSQL will reject all data that does not meet the requirements of the WHERE clause. Because of the dynamic nature of views, the view will only contain rows corresponding to current stock items if you change the **stock** table by adding rows corresponding to new stock items or deleting old ones.

By extending the WHERE clause, this example can be expanded to cover very general data-constraint needs.

Outer Joins

An outer join between two tables treats the two tables unsymmetrically. One of the tables is dominant (often referred to as "preserved"), and the other table is subservient. If the subservient table has no rows satisfying the join condition, the outer join attaches a row of NULL values to the row of the dominant table before projecting the desired columns. To illustrate, let **a** be a column from **tab1** and **b** a column in **tab2**. Further, let the values in the two tables be as shown in the following display:

tab1.a	**tab2.b**
2	4
3	2
5	6
	5

RDSQL syntax requires that the subservient table in an outer join be preceded by the keyword OUTER in the FROM clause. The following SELECT statement contains an outer join between **tab1** (the dominant table) and **tab2** (the subservient table):

```
select a, b
    from tab1, outer tab2
    where a = b
```

The resulting table has the following two columns:

a	**b**
2	2
3	-
5	5

Every value for **a** is present and only those values of **b** that match those in **a**. When there is no value in column **b** that satisfies the join condition, a NULL value (shown here as -) is substituted.

A WHERE clause is required in the case of outer joins and must set a condition between the two tables.

See Chapter 7 and Appendix L for more information about outer joins.

Table Access by Row ID

The keyword ROWID can be used in RDSQL statements to refer to the C-ISAM record number associated with a row in a database table. ROWID can be thought of as a hidden column in every table. When you refer to *table.**, the implied list of columns does *not* include ROWID.
On the other hand, you can use the syntax

```
SELECT ROWID, * FROM table
```

to get the ROWID value for each row. You may also determine the ROWID of the last row that RDSQL dealt with by examining the SQLCA record. See the next section for how to do this.

ROWID can also be used in WHERE clauses to select rows based on their C-ISAM record number. This feature is useful when there is no other unique column in a table.

If a row is deleted from the table, its ROWID may be assigned to a new row. You should not attribute chronological or other significance to the sequential values of ROWID.

SQLCA Record

Proper database management requires that all logical sequences of statements that modify the database continue successfully to completion. If, for example, you UPDATE a customer account to show a reduction of $100 in the payable balance and the next step, to UPDATE the cash balance, fails for some reason, your books will be out of balance. It is prudent to check that every RDSQL statement executes as you anticipated.

INFORMIX-4GL provides two ways to do this: the global variable **status** that indicates errors both from form-related statements and RDSQL statements and a global record SQLCA that allows you to test the success of RDSQL statements. **status** provides the primary information; SQLCA provides additional information.

RDSQL returns a result code into the SQLCA record after executing every RDSQL statement except DECLARE. This record is shown here:

```
DEFINE SQLCA RECORD
        SQLCODE      INTEGER,
        SQLERRM      CHAR(71),
        SQLERRP      CHAR(8),
        SQLERRD      ARRAY[6] OF INTEGER,
        SQLAWARN     CHAR(8)
END RECORD
```

SQLCODE indicates the result of executing an RDSQL statement. It is set to zero for a successful execution of most statements and to NOT-FOUND (= 100) for a successfully executed query that returns zero rows or for a FETCH that seeks beyond the end of an active set.

SQLCODE is negative for an unsuccessful execution.

INFORMIX-4GL sets the global variable **status** equal to SQLCODE after each RDSQL statement. See the section "Error Messages" after the appendixes for the error codes.

SQLERRM	not used at this time
SQLERRP	not used at this time
SQLERRD	an array of six variables of type INTEGER

SQLERRD[1]	not used at this time
SQLERRD[2]	SERIAL value returned or ISAM error code
SQLERRD[3]	the number of rows processed
SQLERRD[4]	the estimated CPU cost for query
SQLERRD[5]	offset of error into the RDSQL statement
SQLERRD[6]	ROWID of last row

SQLAWARN	is a character string of length eight whose individual characters signal various warning conditions (as opposed to errors) following the execution of an RDSQL statement. The characters are blank if no problems were detected.

SQLAWARN[1]	is set to W if one or more of the other warning characters has been set to W. If SQLAWARN[1] is blank, you do not have to check the remaining warning characters.

SQLAWARN[2]	is set to W if one or more data items was truncated to fit into a CHAR program variable.
SQLAWARN[3]	is set to W if an aggregate function (SUM, AVG, MAX, or MIN) encountered a NULL value in its evaluation.
SQLAWARN[4]	is set to W when the number of items in the *select-list* of a SELECT clause is not the same as the number of program variables in the INTO clause. The number of values returned by **RDSQL** is the smaller of these two numbers.
SQLAWARN[5]	is not used at present.
SQLAWARN[6]	is not used at present.
SQLAWARN[7]	is not used at present.
SQLAWARN[8]	is not used at present.

TODAY and USER Functions

RDSQL provides functions to allow you to include the date and the user's name in a statement. TODAY always returns the system date. USER returns a string containing the current user's name. On UNIX systems, this is the login name; on multi-user DOS systems, this is the machine name. The USER function returns the constant **pcuser** on single-user DOS systems.

You can use these functions wherever you use a constant. For example, if you wish to retrieve the rows that you have inserted into a table, you must define a column to contain the USER name. When you insert into the table, use the USER function as follows:

```
INSERT INTO table VALUES (...,USER,...)
```

You can then retrieve the rows that you entered with a select statement, as follows:

```
SELECT * FROM table
   WHERE user_col = USER
```

(See the section "Cursor Management" earlier in this chapter for a discussion on using the SELECT statement to return multiple rows.)

Use the TODAY function in the same way. You can insert the system date into a table with the following statement:

```
INSERT INTO table VALUES (..., TODAY, ...)
```

Use the next statement to retrieve all rows with today's date from a table:

```
SELECT * FROM table
   WHERE date_col = TODAY
```

Chapter 3

Form Building and Compiling

Chapter 3 Table of Contents

Chapter Overview 5

Structure of a Form Specification File 6
  Database Section 7
  Screen Section 8
    Page Layout 8
    Display Fields 10
  Tables Section 12
  Attributes Section 14
    Fields Linked to Database Columns 15
    Form-Only Fields 16
    Multiple-Column Fields 19
    Attributes Syntax 20
      AUTONEXT 21
      COMMENTS 23
      DEFAULT 25
      DISPLAY LIKE 27
      DOWNSHIFT 29
      FORMAT 30
      INCLUDE 33
      NOENTRY 35
      PICTURE 37
      REQUIRED 39
      REVERSE 41
      UPSHIFT 42
      VALIDATE LIKE 44
      VERIFY 46
  Instructions Section 47
    DELIMITERS 47
    Screen Records 49
    Screen Arrays 51

Default Screen Attributes 53

Creating and Compiling a Form 57
  Through the Programmer's Environment 57
  Through the Operating System 58

Using PERFORM Forms in INFORMIX-4GL 60

Chapter Overview

Before you can use a customized screen form in your INFORMIX-4GL program, you must create a form specification file and use FORM4GL to compile this file. The form specification file contains the screen format and the instructions to INFORMIX-4GL about how to display the data.

This chapter describes the syntax and compilation procedure for form specification files. It also describes the **syscolval** and **syscolatt** tables into which you can insert default attributes, formats, and values for your screen fields.

Note: The FORM4GL syntax for forms that you design to work with INFORMIX-4GL is different in several significant ways from the syntax you use to design a screen for the screen transaction program PERFORM, in INFORMIX-SQL. PERFORM forms can be used with INFORMIX-4GL, but you must recompile them using FORM4GL. (See the section "Creating and Compiling a Form" later in this Chapter for information about using FORM4GL.) In addition, not all the PERFORM features are operative. The section "Using PERFORM Forms in INFORMIX-4GL" at the end of this chapter describes the effect of using forms designed for PERFORM in INFORMIX-4GL programs.

Structure of a Form Specification File

Form specification files consist of three required sections (Database, Screen, and Attributes) and two optional sections (Tables and Instructions):

- **Database Section**: Each form specification file must begin with a Database section identifying the database (if any) on which you want to base 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.
- **Tables Section**: Each form specification file must contain a Tables section following the Screen section if you define any field with the name of a column. The Tables section identifies the tables whose columns will determine the fields that appear in the form.
- **Attributes Section**: The Attributes section describes each field on the form—including, for example, appearance, acceptable input values, on-screen comments, and default values.
- **Instructions Section**: The Instructions section is optional and specifies alternate field delimiters and defines screen records and screen arrays.

The following figure illustrates the overall structure of form specification files:

```
DATABASE stores

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

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

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

TABLES
customer orders items manufact

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

INSTRUCTIONS
SCREEN RECORD sc_order[5] (orders.order_date THRU orders.paid_date)
```

Figure 3-1. Schematic for a Form File

Database Section

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

The Database section consists of the DATABASE keyword followed by the name of the database. The structure of the DATABASE section is as follows:

DATABASE
    database-name [WITHOUT NULL INPUT]

Notes

1. The WITHOUT NULL INPUT option should be used only if you have elected to create and work with a database that does not have NULL values (see Appendix I for the other required steps). The effect of this option is to cause INFORMIX-4GL to display as defaults zeros for numeric and DATE fields and blanks for CHAR fields that have no other defaults.

2. It is possible to create a form that is not related to a database. To do so, give the database name as **formonly** and omit the Tables section of the form specification. Use the table name **formonly** in the Attributes section in naming the fields.

Examples

```
database
    stores
```

Screen Section

The Screen section of the form specification file describes how the form will appear on the screen when you use it with INFORMIX-4GL. It begins with the keyword SCREEN and is optionally terminated with the END keyword.

Page Layout

A page layout consists of an array of *display fields* and textual information such as titles and field labels.

Syntax

```
SCREEN
{
      .                                     \
      .                                      |
      .                                      |
display fields and text                      >  page layout
      .                                      |
      .                                      |
      .                                     /
}
[END]
```

Explanation

SCREEN	is a required keyword.
{ }	are required symbols indicating the beginning and ending of the page layout.
END	is an optional keyword.

Notes

1. Do not include more than twenty screen lines between the pair of braces. If you do, **FORM4GL** will split the page into two, with the twenty-first line at the top of the second page. You cannot use multiple-screen forms with **INFORMIX-4GL**.

2. Form specification files based on earlier RDS products use the END keyword in the Screen section. The *INFORMIX-4GL User Guide* follows this practice.

Display Fields

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 section.

Syntax

```
[fieldtag        ]
```

Explanation

[ ]	are delimiters for a field. The width of the field is the number of characters that can be placed between the brackets. In this context, the brackets are required and do not signify an optional syntax.
fieldtag	is the field tag used to identify the display field.

Notes

1. Each field must have a field tag.

2. The field tag is from one to fifty characters long. The first character must be a letter; the rest of the tag may include letters, numbers, and underscores (_). The field tag must fit within the brackets.

3. The same field tag can be used at more than one position in the Screen section of the form specification if you want the same column information to appear at more than one place or if you define a screen array (see "Instructions Section").

4. Field tags are labels and are not the same as field names. The Attributes section links each field tag to a field name.

5. FORM4GL ignores the case of a field tag; **a1** and **A1** are the same.

6. One-character columns are given the display tags **a** through **z**. This means that a screen form can use no more than twenty-six one-character columns.

7. When you create a default form specification file (see "Creating and Compiling a Form"), the widths of all fields are determined by the data type of the corresponding columns in the database tables.

8. If you create your own form, you will normally want the width of a field in the screen section of the form specification to be equal to the width of the program variable or column to which it corresponds.

9. 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 that you assign, INFORMIX-4GL fills the field with asterisks.

10. Fields intended to display CHAR type data can be shorter than a column's defined length. INFORMIX-4GL will fill the field from the left and truncate the right end of any CHAR string that is longer than the field to which it is assigned. Through subscripting, you may assign portions of a CHAR column to one or more fields (see "Attributes Section," later in this chapter).

11. If you edit and modify the default form specification file or create one from scratch, you can verify that the field widths match the width requirements of the corresponding CHAR columns by using the **−v** option of FORM4GL. In response to the system prompt, enter

    ```
    form4gl −v form-name
    ```

    FORM4GL reports any discrepancies in the file **form-name.err**.

Examples

The following is a screen layout from the **orderform.per** form specification file in the INFORMIX-4GL demonstration program.

```
SCREEN
{
-------------------------------------------------------------------------------
                               ORDER FORM
-------------------------------------------------------------------------------
Customer Number:[f000        ]  Contact Name:[f001          ][f002           ]
   Company Name:[f003                  ]
        Address:[f004                  ][f005               ]
           City:[f006              ] State:[a0] Zip Code:[f007 ]
      Telephone:[f008                ]
-------------------------------------------------------------------------------
Order No:[f009        ] Order Date:[f010       ] Purchase Order No:[f011       ]
    Shipping Instructions:[f012                                    ]
-------------------------------------------------------------------------------
Item No.  Stock No.  Code    Description      Quantity    Price        Total
[f013  ]  [f014  ]  [a1 ] [f015           ] [f016   ] [f017      ] [f018        ]
[f013  ]  [f014  ]  [a1 ] [f015           ] [f016   ] [f017      ] [f018        ]
[f013  ]  [f014  ]  [a1 ] [f015           ] [f016   ] [f017      ] [f018        ]
[f013  ]  [f014  ]  [a1 ] [f015           ] [f016   ] [f017      ] [f018        ]
                Running Total including Tax and Shipping Charges:[f019        ]
===============================================================================
}
END
```

Tables Section

Overview

The third section of the form specification file lists all the tables whose columns appear in the screen form. You need not display every column of every table listed.

The Tables section consists of the TABLES keyword, followed by a table name or a list of table names from the database named in the Database section. You may list up to fourteen tables separated by spaces or commas. The structure of the TABLES section is shown below.

```
TABLES
 table-name
 ...
[END]
```

Notes

1. The END keyword is optional.

2. You do not need to include a Tables section if you have specified FORMONLY in the Database section of a form specification file.

Examples

The Tables section from the **orderform** form specification file in Appendix A is as follows:

```
TABLES
     customer
     orders
     items
     stock
```

Attributes Section

Overview

The Attributes section describes the behavior and appearance of each field in the Screen section and associates each field with a field name. Every field in the Screen section must be described in the Attributes section. You use *attributes* to describe how INFORMIX-4GL should display the field, to specify a default value, to limit the values that may be entered, and to set other parameters as described later in the section "Attributes Syntax." The general format of the Attributes Section is

```
ATTRIBUTES
    field-tag = field-description;
    ...
[END]
```

Notes

1. The order in which the fields are described in the Attributes section determines the order for the fields in the default screen records (see "Instructions Section").

2. The END keyword is optional; however, earlier RDS products required it. The *INFORMIX-4GL User Guide* follows the earlier format.

3. Fields in the Screen section do not have to be associated with a column from the database. A field not associated with a column is called a *form-only field.*

The Attributes section contains two kinds of field descriptions: those linking field tags to database columns and those linking field tags to form-only fields.

Fields Linked to Database Columns

Screen fields are associated with database columns only during the compilation of the form specification file. During the compilation process, FORM4GL examines two optional tables, **syscolval** and **syscolatt**, for default values of the attributes that you have associated with columns of the database (see the section "Default Screen Attributes," later in this chapter, for a discussion of these tables). After extracting these default attributes and obtaining the data type from the system catalogs, the association between the fields and database columns is broken. You must mediate between screen fields and database columns with program variables.

Syntax

field-tag = [*table.*]*column*[, *attr-list*];

Explanation

field-tag	is a field tag that identifies a field in the Screen section.
table	is the name of a database table listed in the Tables Section.
column	is the name of a column in *table* or, if *table* is not given, in one of the tables listed in the Tables section.
attr-list	is one or more FORM4GL attributes, separated by commas.

Notes

1. Only the field tag, the equal sign, the column name, and the semicolon are required.

2. You need to specify *table* only if *column* occurs in more than one table in the form. FORM4GL issues an error during compilation if there is ambiguity. Because you will often refer to field names collectively through a screen record (see "Instructions Section") built upon all the fields related to a single table, you may find your specification files easier to work with if you specify *table* for each field.

3. You can display a portion of a CHAR type column 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 the first portion of **ship_instruct** on the screen as a field of length 20. Because you cannot use subscripts in the field-name list in an INFORMIX-4GL INPUT, INPUT ARRAY, DISPLAY, or DISPLAY ARRAY statement, you must use form-only fields (described in the next section) to handle the second portion in data entry and display. If the field tags for the two fields are **inst1** and **inst2**, respectively, the Attributes section entry is as follows:

   ```
   inst1 = orders.ship_instruct[1,20];
   inst2 = FORMONLY.ship_in2;
   ```

   In this case, the default screen record **orders.*** does not have a component corresponding to **inst2**.

Form-Only Fields

Form-only fields are not associated with columns of the database. They can be used to display information on the screen and to retrieve data from the screen.

Syntax

fieldtag = FORMONLY.*field-name*
  [TYPE [*data-type* | LIKE *table.column*]][NOT NULL] [, *attr-list*];

Explanation

field-tag	is the field tag used in the SCREEN section.
FORMONLY	is the keyword indicating that the field does not correspond to a column of a table in the database.
field-name	is an RDSQL identifier for the name of the field.
TYPE	is an optional keyword.
data-type	is any one of the data types permitted by RDSQL except SERIAL (see Chapter 2 for the definition of the data types).
LIKE	is an optional keyword.
table.column	is the name of a column in the database.
NOT NULL	are optional keywords informing INFORMIX-4GL that, if you name this field in an INPUT or INPUT ARRAY statement, the user must give it a value.
attr-list	is a list of one or more attributes separated by commas.

Notes

1. You must give a value for *data-type* only when you want to use the INCLUDE or DEFAULT attribute for this entry. Otherwise, FORM4GL assumes the field is a CHAR field whose length is the width of the field. INFORMIX-4GL performs the necessary data conversion for the corresponding program variable during input or display.

2. When describing *data-type*, do not give a length to type CHAR, DECIMAL, or MONEY; the length is determined by the display width.

3. When you specify one or more form-only fields, INFORMIX-4GL behaves as though these fields formed a database table named **formonly**, with the field names as column names.

4. When the Database section has the WITHOUT NULL INPUT clause, the NOT NULL keyword instructs INFORMIX-4GL to use zero (numeric type) or blanks (CHAR type) as a default for this field in INPUT or INPUT ARRAY statements. If you do not specify the type, INFORMIX-4GL treats the field as type CHAR.

Examples

The following form-only fields could be used in an order entry form to display information about items:

```
f020 = formonly.manu_name;
f021 = formonly.description;
f022 = formonly.unit_price;
f023 = formonly.unit_descr;
```

The demonstration application uses the following form-only field to store the running total price for the order as items are entered:

```
f019 = formonly.t_price
```

Multiple-Column Fields

A screen form that contains information from several database tables may include screen fields that you will use to display data via program variables from two (or more) database columns.

The database columns you assign to the same field must have the same field size. Usually, they will have the same data type. If they are CHAR columns, they must have the same length.

You assign columns to the same field tag in the Attributes section:

---

field-tag = *table1.column* = *table2.column*;

The placement of attributes determines when they take effect. When **INFORMIX-4GL** executes an INPUT, INPUT ARRAY, DISPLAY, or DISPLAY ARRAY statement, the screen fields listed (explicitly or implicitly) in the statement are called *active* fields. If you want an attribute to apply regardless of which field name is active, place the attribute after the last field name:

---

field-tag = *table1.column* = *table2.column*, *attr-list*;

If you want different attributes to apply for each of the field names, place a semicolon after the attribute list for the first field name:

```
field-tag = table1.column, attr-list1;
  = table2.column, attr-list2;
```

attr-list1 is effective when *table1.column* is active, and *attr-list2* is effective when *table2.column* is active.

The FORMAT and REVERSE attributes, described later in this chapter, always take effect, regardless of their placement.

Attributes Syntax

FORM4GL recognizes the attributes listed below. The syntax for each attribute is detailed in the following sections.

AUTONEXT	NOENTRY
COMMENTS	PICTURE
DEFAULT	REQUIRED
DISPLAY LIKE	REVERSE
DOWNSHIFT	UPSHIFT
FORMAT	VALIDATE LIKE
INCLUDE	VERIFY

AUTONEXT

Overview

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

Syntax

---

field-tag = *table.column*, AUTONEXT;

Explanation

field-tag	is the field tag used in the Screen section.
table.column	is the name of a field (either related to a column or form-only).
AUTONEXT	is a keyword that tells **INFORMIX-4GL** to advance the cursor to the next field when the current field is full.

Notes

1. You specify the order of fields in each INPUT or INPUT ARRAY statement.

2. AUTONEXT is particularly useful with CHAR fields in which the input data is of a standard length (a zip code column always requires five digits, and the abbreviation for a state is always two letters), 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).

3. If the data entered into the field does not meet the requirements of other attributes like INCLUDE or PICTURE, the cursor does not automatically move to the next field, but remains in the current field.

Examples

The demonstration application uses the **customer** form to enter all the names and addresses of the customers. The following excerpt from the Attributes section of the **customer** form uses the AUTONEXT attribute:

```
  . . .
  a0 = customer.state, DEFAULT = "CA", AUTONEXT;
  f007 = customer.zipcode, AUTONEXT;
  f008 = customer.phone;
  . . .
```

When two characters are entered into the **customer.state** field (thus filling the field), the cursor moves automatically to the beginning of the next field (the **customer.zipcode** field). When five characters are entered into the **customer.zipcode** field (filling this field), the cursor moves automatically to the beginning of the next field (the **customer.phone** field).

COMMENTS

Overview

Use COMMENTS to cause INFORMIX-4GL to display a message on the Comment Line at the bottom of the screen. The message is displayed when the cursor moves to the associated field and is erased when the cursor moves to another field.

Syntax

field-tag = *table.column*, COMMENTS = "*message*";

Explanation

field-tag	is the field tag used in the Screen section.
table.column	is the name of a field (either related to a column or form-only).
COMMENTS	is a required keyword.
message	is a character string enclosed in quotation marks.

Notes

1. The *message* must appear in quotation marks on a single line of the form specification file.

2. The default position of the Comment Line on the screen is line 23. You can reset this position with the OPTIONS statement.

3. The default position of the Comment Line in a window is LAST. You can reset this position in the OPTIONS statement (if you want the new position in all windows) or in the

ATTRIBUTE clause of the appropriate OPEN WINDOW statement (if you want the new position in a specific window).

4. The most common application 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 values.

Examples

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

DEFAULT

Overview

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

Syntax

---

field-tag = *table.column*, DEFAULT = *value*;

Explanation

field-tag	is the field tag used in the Screen section.
table.column	is the name of a field (either related to a column or form-only).
DEFAULT	is a required keyword.
value	is the default value.

Notes

1. Default values have no effect when you execute the INPUT statement using the WITHOUT DEFAULTS option. In this case, **INFORMIX-4GL** displays the values in the program variables list on the screen. The situation is the same for the INPUT ARRAY statement except that **INFORMIX-4GL** displays the default values when you insert a new row.

2. If you use the WITHOUT NULL INPUT option in the Database section and you do not use the DEFAULT attribute, CHAR and DATE fields default to blanks, numeric fields to 0, and money fields to $0.00.

3. If you do not use the WITHOUT NULL INPUT option in the Database section, all fields default to NULL values unless you use the DEFAULT attribute.

4. If the field type is CHAR or DATE, enclose *value* in quotation marks.

5. If both the DEFAULT attribute and the REQUIRED attribute are assigned to the same field, the REQUIRED attribute is ignored.

6. Use the TODAY keyword as the *value* to assign the current date as the default value of a field.

Examples

```
c8 = state, UPSHIFT, AUTONEXT,
        DEFAULT = "CA";

o12 = order_date, DEFAULT = TODAY;
```

DISPLAY LIKE

Overview

Use the DISPLAY LIKE attribute to cause **INFORMIX-4GL** to display the field using the attributes assigned to a database column in the **syscolatt** table.

Syntax

---

field-tag = *table.column*, DISPLAY LIKE *tbl.col*;

Explanation

field-tag	is the field tag used in the Screen section.
table.column	is the name of a field (either related to a column or form-only).
DISPLAY LIKE	are required keywords.
tbl.col	is the name of a column in the database.

Notes

1. This attribute is equivalent to listing all the attributes that you have assigned to *tbl.col* in the **syscolatt** table. See the section "Default Screen Attributes" for details on the **syscolatt** table.

2. You do not need the DISPLAY LIKE attribute if *table.column* is the same as *tbl.col*.

Examples

```
s12 = formonly.total, DISPLAY LIKE items.total_price
```

Related Attributes

VALIDATE LIKE

DOWNSHIFT

Overview

Assign the DOWNSHIFT attribute to a CHAR field when you want **INFORMIX-4GL** to convert uppercase letters entered by the user to lowercase letters, both on the screen and in the corresponding program variable.

Syntax

```
field-tag = table.column, DOWNSHIFT;
```

Explanation

field-tag	is the field tag used in the SCREEN section.
table.column	is the name of a database column.
DOWNSHIFT	is the keyword that instructs **INFORMIX-4GL** to convert CHAR input data to lowercase letters in the program variable.

Notes

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

Related Attributes

UPSHIFT

FORMAT

Overview

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

Syntax

---

field-tag = *table.column*, FORMAT = "*format-string*";

Explanation

field-tag	is the field tag used in the Screen section.
table.column	is the name of a field (either related to a column or form-only).
FORMAT	is a required keyword.
format-string	is a string of characters that specifies the desired data format. You must enclose *format-string* in quotation marks.

Notes

1. For DECIMAL, SMALLFLOAT, or FLOAT data types, *format-string* consists of pound signs (#) that represent digits and a decimal point. For example, "###.##" produces at least three places to the left of the decimal point and exactly two to the right.

2. If the actual displayed number is shorter than the *format-string*, INFORMIX-4GL right-justifies it and pads the left with blanks.

3. If the *format-string* is smaller than the display width, FORM4GL gives a warning, but the form is usable. INFORMIX-4GL displays the data right-justified in the field.

4. If necessary to satisfy the format, INFORMIX-4GL rounds numbers before displaying them.

5. For DATE data types, INFORMIX-4GL recognizes the following symbols as special in the string *format-string*:

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, FORM4GL interprets any other characters as literals and displays them wherever you place them within *format-string*.

Examples

For DATE fields:

Input	Result
no FORMAT attribute	09/15/1986
FORMAT = "mm/dd/yy"	09/15/86
FORMAT = "mmm dd, yyyy"	Sep 15, 1986
FORMAT = "yymmdd"	860915
FORMAT = "dd-mm-yy"	15-09-86
FORMAT = "(ddd.) mmm. dd, yyyy"	(Sat.) Sep. 15, 1986

Related Attributes

PICTURE

INCLUDE

Overview

Use the INCLUDE attribute to specify acceptable values for a field and to cause **INFORMIX-4GL** to check before accepting an input value.

Syntax

field-tag = *table.column*, INCLUDE = (*value-list*);

Explanation

field-tag	is the field tag used in the Screen section.
table.column	is the name of a field (either related to a column or form-only).
INCLUDE	is a required keyword.
value-list	is either a list of individual values (*value1*, *value2*, ...) or a range of values (*value1* TO *value2*) or a combination of individual values and ranges separated by commas.

Notes

1. When you specify a range of values, the lower value must appear first.

2. For ranges of character values, INFORMIX-4GL uses dictionary ordering with the printable ASCII character set (see Appendix M for an ordered list of the ASCII character set). In a numeric 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.

3. If you include a character string that contains a blank space, a comma, or any special characters, or does not begin with a letter, you must enclose the entire string in quotation marks. It is advisable to enclose character strings in quotation marks at all times.

4. Before INFORMIX-4GL accepts a new row, you must enter an acceptable value in each display field with the INCLUDE attribute. If the list of acceptable values in the *value-list* does not include the default value, the INCLUDE attribute behaves like the REQUIRED attribute, and an acceptable entry is required.

5. Including a COMMENTS attribute indicating acceptable values makes data entry easier.

Examples

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

Related Attributes

COMMENTS

NOENTRY

Overview

Use the NOENTRY attribute to prevent data entry during an INPUT or INPUT ARRAY statement.

Syntax

field-tag = *table.column*, NOENTRY;

Explanation

field-tag	is the field tag used in the Screen section.
table.column	is the name of a field (either related to a column or form-only).
NOENTRY	is the keyword indicating that no data entry may be made to this field during an INPUT or INPUT ARRAY statement.

Notes

1. The NOENTRY attribute does not prevent you from entering data into the field during a CONSTRUCT statement (to create a query-by-example).

Examples

```
i13 = items.stock_num;
    = stock.stock_num, NOENTRY;
```

When you are entering data intended for the **stock** table, the **stock_num** column is not available, since this SERIAL column gets its value from RDSQL during the INSERT statement. You can, however, use the same field to enter the stock number intended for the **items** table.

PICTURE

Overview

Use the PICTURE attribute to specify the character pattern for data entry to a CHAR type field.

Syntax

---

field-tag = *table.column*, PICTURE = "*format-string*";

Explanation

field-tag	is the field tag used in the Screen section.
table.column	is the name of a field (either related to a column or form-only).
PICTURE	is a required keyword.
format-string	is a string of characters that specify the desired character pattern.

Notes

1. *format-string* is a combination of three special symbols:

Symbol	**Meaning**
A	Any letter
#	Any digit
X	Any character

INFORMIX-4GL treats any other character in the *format-string* as a literal. The cursor skips over such literal characters during data entry.

2. INFORMIX-4GL displays the literal characters in the display field and leaves blanks elsewhere.

3. If you attempt to enter a character not in conformity with the *format-string*, your terminal beeps and INFORMIX-4GL does not echo the character on the screen.

4. The PICTURE attribute does not require the entry of the entire field; it only requires that what you enter conforms to *format-string*.

5. The PICTURE must fill the entire width of the display field.

Examples

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

produces the following display field before data entry:

```
[   -   -    x     ]
```

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

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

INFORMIX-4GL accepts any of the following inputs:

```
LF49367-BB(*)
TG38524-AS(3)
YG67489-ZZ(D)
```

The user does not enter the "-" nor the parentheses, but INFORMIX-4GL includes them in the string passed to the program variable.

REQUIRED

Overview

Use the REQUIRED attribute to force data entry in a particular field during an INPUT or INPUT ARRAY statement.

Syntax

---

field-tag = *table.column*, REQUIRED;

Explanation

field-tag	is the field tag used in the Screen section.
table.column	is the name of a field (either related to a column or form-only).
REQUIRED	is the keyword that instructs **INFORMIX-4GL** to insist upon data entry to the *field-tag* field.

Notes

1. The REQUIRED keyword is effective only when *table.column* occurs in the list of screen fields of an INPUT or INPUT ARRAY statement.

2. There is no default value for a REQUIRED field. If you assign both the REQUIRED attribute and the DEFAULT attribute to the same field, **INFORMIX-4GL** assumes that the DEFAULT value satisfies the REQUIRED attribute.

3. The REQUIRED attribute requires only that the user enter a printable character in the field. If the user subsequently erases the entry during the same input, INFORMIX-4GL considers the REQUIRED attribute satisfied. If you want to insist on a non-NULL entry, make the field form-only and NOT NULL.

Examples

```
o20 = orders.po_num, REQUIRED;
```

INFORMIX-4GL requires the entry of a purchase order value when you collect information for a new order.

REVERSE

Overview

Assign the REVERSE attribute to the fields you want INFORMIX-4GL to display in reverse video.

Syntax

field-tag = *table.column*, REVERSE;

Explanation

field-tag	is the field tag used in the Screen section.
table.column	is the name of a field (either related to a column or form-only).
REVERSE	is the keyword that instructs **INFORMIX-4GL** to display the *field-tag* field in reverse video.

Notes

1. On terminals that do not support reverse video, fields having the REVERSE attribute are enclosed in angle brackets < >.

UPSHIFT

Overview

Assign the UPSHIFT attribute to a CHAR field when you want INFORMIX-4GL to convert lowercase letters in data entry to uppercase letters, both on the screen and in the program variable corresponding to that field.

Syntax

field-tag = *table.column*, UPSHIFT;

Explanation

field-tag	is the field tag used in the Screen section.
table.column	is the name of a field (either related to a column or form-only).
UPSHIFT	is the keyword that instructs INFORMIX-4GL to convert CHAR input data to uppercase.

Notes

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

Examples

```
c8   = state, UPSHIFT, AUTONEXT,
       INCLUDE = ("CA", "OR", "NV", "WA"),
       DEFAULT = "CA" ;
```

Because of the UPSHIFT attribute, INFORMIX-4GL enters uppercase characters in the **state** field regardless of the case used to enter them.

The AUTONEXT attribute tells INFORMIX-4GL to move automatically to the next field once you type the total number of characters allowed for the field (in this instance, two characters). The INCLUDE attribute restricts entry in this field to the characters CA, OR, NV, or WA only. The DEFAULT value for the field is CA.

Related Attributes

DOWNSHIFT

VALIDATE LIKE

Overview

Use the VALIDATE LIKE attribute to cause **INFORMIX-4GL** to validate the data entered into the field, using the attributes assigned to a database column in the **syscolval** table.

Syntax

field-tag = *table.column*, VALIDATE LIKE *tbl.col*;

Explanation

field-tag	is the field tag used in the Screen section.
table.column	is the name of a field (either related to a column or form-only).
VALIDATE LIKE	are required keywords.
tbl.col	is the name of a column in the database.

Notes

1. This attribute is equivalent to listing all the attributes that you have assigned to *tbl.col* in the **syscolval** table. See the section "Default Screen Attributes," later in this chapter, for details on the **syscolval** table.

2. You do not need the VALIDATE LIKE attribute if *table.column* is the same as *tbl.col.*

Examples

```
s13 = formonly.state, VALIDATE LIKE customer.state;
```

Related Attributes

DISPLAY LIKE

VERIFY

Overview

Use the VERIFY attribute when you want **INFORMIX-4GL** to require users to enter data twice for a particular field in order to reduce the probability of erroneous data entry.

Syntax

---

field-tag = *table.column*, VERIFY;

Explanation

field-tag	is the field tag used in the Screen section.
table.column	is the name of a field (either related to a column or form-only).
VERIFY	is the keyword that instructs **INFORMIX-4GL** to require duplicate data entry to the *field-tag* field.

Notes

1. Since 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**, **INFORMIX-4GL** erases the field and requests that you re-enter the value. You must enter *exactly* the same data each time, character for character: 15000 is not exactly the same as 15000.00.

Examples

For example, if you specify a field for salary information like this:

```
s10 = quantity, VERIFY;
```

INFORMIX-4GL requires the entry of exactly the same data twice.

Instructions Section

The final section of the form specification file is the optional Instructions section. This section is used to

- Specify alternate field delimiters
- Define screen records and arrays

The Instructions section begins with the INSTRUCTIONS keyword. It appears after the last attribute line of the Attributes section or the optional END keyword that concludes the Attributes section.

```
INSTRUCTIONS
   Instructions
   ...
[END]
```

The END keyword is optional and may be omitted.

DELIMITERS

You may change the delimiters that INFORMIX-4GL uses to enclose fields when the form appears on the screen from brackets [ ] to any other printable character, including blank spaces.

Syntax

DELIMITERS "*ab*"

Explanation

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

Notes

1. The DELIMITERS instruction tells INFORMIX-4GL what symbols to use as delimiters when it displays the fields on the screen.

2. FORM4GL still requires brackets in the Screen section of a form specification file.

3. Because the delimiter takes up a space, the closest that two fields can be on the screen is, ordinarily, two spaces. If you want to design your form to have only one space between fields, the following rules apply:

   - In the Screen section of your form specification file, use the vertical bar (|) to separate two fields in place of the standard pair of back-to-back brackets][.

```
SCREEN
{
  . . .
  Full Name-[f011          |f012          ]
  . . .
}
```

- In the Instructions section, define two new delimiters using identical symbols for both the beginning and ending delimiters (for example, “||” or “//” or “::”).

  ```
  DELIMITERS "::"
  ```

  When the fields **f011** and **f012** are displayed on the screen, they will appear as:

  ```
  Full Name-:               :              :
  ```

Screen Records

You can collect groups of screen fields into screen records. You define the screen records in the Instructions section of the form specification file and refer to them in your **INFORMIX-4GL** program.

Syntax

SCREEN RECORD *record-name*
  ({*table-name.** |
  table-name.column1 THRU *table-name.column2* |
  table-name.column}[, ...])

Explanation

SCREEN RECORD	are required keywords.
record-name	is an **RDSQL** identifier.
table-name	is the name of a table (or the keyword FORMONLY) that you have used in the Attributes section to define a field name.

column1, *column2*, *column*	are field names.
THRU	is an optional keyword. The keyword THROUGH is a synonym.

Notes

1. *table-name* can be FORMONLY as well as a database table name.

2. FORM4GL creates a default screen record for each table in the form specification file that is used to identify a field. The default record, which has the name of the table, contains components corresponding to only those columns in the table that are fields on the form. The order of the components is the order of the field names in the Attributes section. It is an error to define a screen record in the Instructions section with the name of a table in the form.

3. The option of giving a range of field names with the THRU keyword assumes the order in which the fields are listed in the Attributes section. The THRU keyword is shorthand for listing all fields in the Attributes list between *column1* and *column2*, inclusive of these fields as well.

Examples

The following example assigns a part of the fields corresponding to the columns of the **customer** table to a screen record. Such a screen record simplifies the INFORMIX-4GL statements that update customer address and telephone data.

```
SCREEN RECORD address
      (customer.address1 THRU customer.phone)
```

All of the fields corresponding to columns from the **customer** table on the screen form constitute a default screen record with *record-name* **customer**.

Screen Arrays

You may collect groups of screen fields into screen arrays. A screen array most often will be a physical array of identical rows of fields on the screen. Each column of the screen array consists of fields with the same tag. You define the screen arrays in the Instructions section of the form specification file and refer to them in your **INFORMIX-4GL** program.

Syntax

```
SCREEN RECORD record-name[n]
        ({table-name.* |
        table-name.column1 THRU table-name.column2 |
        table-name.column}[, ...])
```

Explanation

The syntax explanation is the same as for screen records (see the previous section) with the addition that *n* is a required integer parameter enclosed in brackets.

Notes

1. The integer *n* is the number of rows in a screen array.

Examples

To illustrate, consider the following fragment from a form specification file:

```
SCREEN
{
. . .
Item 1 [p      ][q      ][u      ][t         ]
Item 2 [p      ][q      ][u      ][t         ]
Item 3 [p      ][q      ][u      ][t         ]
Item 4 [p      ][q      ][u      ][t         ]
Item 5 [p      ][q      ][u      ][t         ]
}

TABLES orders, items, stock

ATTRIBUTES
. . .
p = stock.stock_num;
q = items.quantity;
u = stock.unit_price;
t = items.total_price;
. . .
INSTRUCTIONS
SCREEN RECORD sc_items[5] (stock.stock_num,
            items.quantity, stock.unit_price,
            items.total_price)
```

The screen array has five rows and four columns. The rows are numbered from 1 to 5.

Assuming there are no other columns of the **items** table in the form, there is a default screen record **items** containing two components: **quantity** and **total_price**.

Default Screen Attributes

As an alternative to entering attributes in the form specification file, you can enter them into two database tables, **syscolval** and **syscolatt**, using the **upscol** utility described in Appendix E. During the compilation of a form, **FORM4GL** searches these tables for data validation and screen attribute information for each column given as a field name. **FORM4GL** adds the attributes listed in these files to the attributes listed in the form specification file. In case of conflict, the attributes explicitly mentioned in the form specification file have precedence. **INFORMIX-4GL** enforces the resulting set of attributes during the execution of the INPUT, INPUT ARRAY (**syscolval**) DISPLAY, and DISPLAY ARRAY (**syscolatt**) statements.

The schema of the tables are as follows:

syscolval

tabname	char(18)
colname	char(18)
attrname	char(10)
attrval	char(64)

syscolatt

tabname	char(18)
colname	char(18)
seqno	serial
color	smallint
inverse	char(1)
underline	char(1)
blink	char(1)
left	char(1)
def_format	char(64)
condition	char(64)

tabname and **colname** are the names of the table and column to which the attributes apply. The permissible values for **attrname** and **attrval** in **syscolval** are shown in the following table:

attrname	**attrval**
INCLUDE	as in this chapter
PICTURE	as in this chapter
DEFAULT	as in this chapter
COMMENTS	as in this chapter
SHIFT	UP, DOWN, NO (the default)
VERIFY	YES, NO (the default)
AUTONEXT	YES, NO (the default)

The **color** column in **syscolatt** describes both color (for terminals that can display color) and intensities (for terminals with monochrome displays). The possible values for **color** are white, yellow, magenta, red, cyan, green, blue, and black. The following table shows the correspondence between the default color names and intensities.

#	**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. 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.

You may change the color names from the default list by associating different numbers with different color names in a file named **colornames**. If it exists in $INFORMIXDIR/incl (see Appendix C), **FORM4GL** examines **colornames** at compile time to obtain the correspondence between the numbers 0 through 7 and the color names. (See Appendix N for the format of the **colornames** file.)

The values for **inverse**, **underline**, and **blink** are Y (yes) and N (no). The default for each of these columns is N, that is, normal display, no underline, and steady font. Which of these attributes can be displayed simultaneously with the color combinations or with each other is terminal-dependent.

The color, intensity, and other screen attributes interact with the **termcap** file on the user's computer for UNIX systems. Appendix N describes the changes that should be made to your **termcap** entry to enable these features for your terminal. On DOS systems, you can set your terminal characteristics at installation.

The **def_format** column takes the same string that you would enter for the FORMAT attribute in a screen form. Do not use quotation marks.

The **condition** column takes string values that are a restricted set of the "simple" WHERE clauses of a SELECT statement, except that the WHERE keyword and the column name are omitted. "Simple" means that compound conditions formed by combining several Boolean expressions with AND or OR are not allowed. Examples of permitted entries for the **condition** column are as follows:

```
<= 100
BETWEEN 101 AND 1000
>= 1001
MATCHES "[A-M]*"
IN ("CA", "OR", "WA")
NOT LIKE "%analyst%"
```

INFORMIX-4GL assumes that the column identified by **tabname** and **colname** is the subject of all comparisons.

The VALIDATE statement checks a program record or variable list against **syscolval**.

The INITIALIZE statement looks up the default values listed in **syscolval** and assigns them to variables.

Following are the screen attributes allowed in ATTRIBUTE clauses of **INFORMIX-4GL** statements:

WHITE = NORMAL	REVERSE
YELLOW = BOLD	BLINK
MAGENTA = BOLD	UNDERLINE
RED = BOLD	
CYAN = DIM	
GREEN = DIM	
BLUE = DIM	
BLACK = INVISIBLE	

On color terminals, NORMAL is interpreted as WHITE; BOLD as RED; DIM as BLUE, and INVISIBLE as BLACK. If you have a **colornames** file, you may also use the color names listed there.

You may override the default attributes in **syscolatt** by using either the ATTRIBUTE clause in a DISPLAY or DISPLAY ARRAY statement or assigning attributes in the form specification file. If you use the ATTRIBUTE clause in a DISPLAY or DISPLAY ARRAY statement, **INFORMIX-4GL** displays only the attributes mentioned in that clause. There is no carry-over of unmentioned display attributes from the compiled form (except FORMAT). For example, if a column is designated as RED and BLINK in **syscolatt** and you use the statement DISPLAY ... ATTRIBUTE BLUE, you get only the attribute BLUE. You do not get blinking BLUE.

As stated earlier, the attributes in a form specification file and those in **syscolval** and **syscolatt** are combined, with the form specification file taking precedence in the event of conflict. The color (or intensity) attributes are available only through the ATTRIBUTE clause or through **syscolatt**, and conditional attributes are available only through **syscolatt**. You cannot use them in a form specification file.

You can use the utility program **upscol** to create and enter attribute information into **syscolval** and **syscolatt** (see Appendix E for a description of **upscol**).

Creating and Compiling a Form

You can create a form specification file and its customized screen form either through the INFORMIX-4GL Programmer's Environment or directly from the operating system. Either alternative requires that you have already created the database and all the tables to which the form refers. The following two subsections describe these alternative procedures.

Through the Programmer's Environment

To create a customized screen form using the Programmer's Environment, you must follow these steps:

1. Enter `i4gl` in response to your operating system prompt.

2. Select **Form** and then **Generate** from the menu. (Alternately, you can select the **New** option. INFORMIX-4GL prompts you for a form name, prompts you for a system editor if you have not already selected one, and places you in that editor with an empty form specification file. You enter all form specification instructions. The **Generate** option is usually a more efficient way to create a custom form.)

3. Enter the name of the database and the name you want to assign to the form (for example, **myform**). INFORMIX-4GL asks you for the names of the tables whose columns you want in your form. When you have selected all the tables you want to include, FORM4GL creates a default form specification file, as well as a compiled default form. INFORMIX-4GL returns you to the FORM Menu.

4. The default form specification file formats the screen as a list of all the columns in the tables you entered in Step 3. It does not provide any special instructions to INFORMIX-4GL about how to display the data. Select the **Modify** option, and INFORMIX-4GL presents the MODIFY FORM Screen. Select the default form specification (given as **myform**

earlier), and INFORMIX-4GL calls a system editor to display the file. Edit the default form specification file to produce your customized screen form and associated instructions. (You can specify an editor using the DBEDIT environment variable. This is fully explained in Appendix C.) When you write your file and quit the editor, you return to the MODIFY FORM Menu.

5. Select **Compile**. If your form specification file compiles successfully, FORM4GL creates a form file with the extension **.frm** (for example, **myform.frm**). Go on to Step 7. If your form specification file does not compile successfully, go on to Step 6.

6. Select the **Correct** option from the COMPILE FORM Menu. INFORMIX-4GL again calls your editor to display the form specification file, with the compilation errors marked. When correcting your errors, you need not delete the error messages. INFORMIX-4GL does that for you. Save the file and go to Step 5.

7. Save your form specification file with the **Save-and-exit** option.

Through the Operating System

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

1. Create a default form specification file by entering the command

   ```
   form4gl —d
   ```

   at the operating system prompt. FORM4GL asks for the name of your form specification file, the name of your database, and the name of a table whose columns you want in your form. It continues to ask for another table name until you enter a **RETURN** for the name of a table. **FORM4GL** then creates a default form specification file and appends the extension **.per** to its name. It also creates a compiled default form with the extension **.frm**.

2. Use a system editor to modify the default form specification file to meet your specifications. If, as an alternative, you create a form specification file from scratch and skip Step 1, be sure to give the filename the extension **.per**.

3. Enter the command

   ```
   form4gl myform
   ```

   where **myform** is the name of your form specification file (without the **.per** extension). If the compilation is successful, **FORM4GL** creates a compiled form file called **myform.frm** and you are finished creating your customized screen form. If not, **FORM4GL** creates a file named **myform.err**, and you need to go on to Step 4.

4. Review the file **myform.err** to discover the compilation errors. Make corrections in the file **myform.per**. Go to Step 3.

Using PERFORM Forms in INFORMIX-4GL

If you have designed forms to use with the PERFORM screen transaction program of INFORMIX-SQL, you need to know how those forms behave when used with INFORMIX-4GL. The following is a list of the features regarding forms that differ between PERFORM and INFORMIX-4GL:

- Only the DELIMITERS statement in the Instructions section of a PERFORM form specification is applicable in INFORMIX-4GL. All other statements are ignored. To achieve the same effects, you must code them into your INFORMIX-4GL program (see the BEFORE and AFTER clauses of the INPUT statement).
- Multiple-screen forms will not work with INFORMIX-4GL and will produce undesirable overlays. You can achieve similar effects in INFORMIX-4GL, but you must code them explicitly.
- There is no concept of "current table" in INFORMIX-4GL. A single INPUT or INPUT ARRAY statement allows you to enter data into fields that correspond to columns in different tables.
- Joins defined in the PERFORM form specification are ignored in INFORMIX-4GL. You may associate two field names to the same field tag using the same notation as in a PERFORM join, but no join is effected. On the other hand, you can create more complex joins and look-ups in INFORMIX-4GL using the full power of RDSQL.
- The PERFORM attributes LOOKUP, NOUPDATE, QUERYCLEAR, RIGHT, and ZEROFILL are inoperative in INFORMIX-4GL.
- The default attributes listed in **syscolval** and **syscolatt** do not apply to your PERFORM forms unless you recompile them.

Chapter 4

Report Writing

Chapter 4 Table of Contents

Chapter Overview 5

Calling a REPORT Routine 6

Structure of a REPORT Routine 7
- DEFINE Section 8
- OUTPUT Section 9
  - REPORT TO 10
  - LEFT MARGIN 12
  - RIGHT MARGIN 14
  - TOP MARGIN 16
  - BOTTOM MARGIN 18
  - PAGE LENGTH 20
- ORDER BY Section 21
- FORMAT Section 22
  - EVERY ROW 24
- Control Blocks 28
  - AFTER GROUP OF 29
  - BEFORE GROUP OF 32
  - FIRST PAGE HEADER 35
  - ON EVERY ROW 37
  - ON LAST ROW 39
  - PAGE HEADER 41
  - PAGE TRAILER 44
- Statements 46
  - NEED 47
  - PAUSE 48
  - PRINT 50
  - PRINT FILE 53
  - SKIP 54
- Expressions and Built-in Functions 56
  - Aggregates 57
  - LINENO 60
  - PAGENO 61
  - TIME 62

Chapter Overview

INFORMIX-4GL provides all the tools of a general-purpose relational report writer. Reports in INFORMIX-4GL have the following features:

- You have full control over page layout for your report. This includes first-page headers that differ from headers on subsequent pages, page trailers, columnar data presentation, special formatting before and after groups of sorted data, and conditional formatting that depends upon the data.

- INFORMIX-4GL provides aggregate functions that permit you to compute percentages, sums, averages, maximums and minimums.

- You can use all the built-in functions of INFORMIX-4GL, including the USING function (see Chapter 1).

- You can create the report either from the rows returned by a cursor or from report records assembled from any other source, such as the output of several different SELECT statements.

- You can manipulate the data returned by the cursor on a row-by-row basis either before or after the row is formatted by the report.

- You can update the database or perform any other sequence of INFORMIX-4GL statements in the middle of writing a report if the intermediate values calculated by the report meet your criteria. For example, you could even write an alert message containing a second report.

You must define a routine of type REPORT outside the MAIN program. This chapter describes the rules for writing REPORT routines.

Calling a REPORT Routine

You call a REPORT routine through three statements that occur before, during, and after a program loop that you define:

START REPORT *report-name*
  [TO {PRINTER | PIPE *program* | *filename*}]

OUTPUT TO REPORT *report-name*(*variable-list*)

FINISH REPORT *report-name*

The basic loop structure (whether a FOR, FOREACH, or WHILE loop) is illustrated below:

```
START REPORT report1
begin loop                # of whatever kind
    . . .
    OUTPUT TO REPORT report1(customer.*)
    . . .
end loop
FINISH REPORT report1
```

The START REPORT statement initializes the report and, optionally, determines the output file or device. The OUTPUT TO REPORT statement tells INFORMIX-4GL to process the next row of the report. The FINISH REPORT statement handles the end of report summaries. You can find the full syntax for these three statements in Chapter 7.

Structure of a REPORT Routine

A report routine is composed of sections that are made up of control blocks and/or statements. Each statement can contain clauses made up of keywords and expressions. You must observe the order of the sections shown in the following syntax diagram when you write an **INFORMIX-4GL** report routine.

Syntax

```
REPORT report-name (argument-list)
     [DEFINE section]
     [OUTPUT section]
     [ORDER BY section]
     FORMAT section
END REPORT
```

REPORT	is a required keyword.
report-name	is an **INFORMIX-4GL** identifier.
argument-list	is a list of variables or record identifiers, separated by commas. Record identifiers must not have a .* extension.
END REPORT	are required keywords.

Notes

1. The DEFINE, OUTPUT, ORDER BY, and FORMAT sections are described in later sections of this chapter.

2. A minimal report consists only of the FORMAT section. You may, optionally, include other sections as needed.

Examples

Several REPORT routines are included with the demonstration application listed in Appendix A. They illustrate a wide variety of the commands available for writing reports with INFORMIX-4GL and provide most of the example text in the pages that follow.

DEFINE Section

An INFORMIX-4GL REPORT routine requires a DEFINE section when you pass arguments to the report or use local variables in the report. You must have an *argument-list* under the following conditions:

- When there is an ORDER BY section in your report. In this case, you must pass all the values for each row of the report.
- When you use the GROUP PERCENT aggregate function anywhere in your report or use any aggregate function over all the rows of the report at any place except in the ON LAST ROW control block. In short, if you print an aggregate dependent on all rows of the report in the middle of the report, you must pass the rows of the report through the *argument-list*.
- When you use the FORMAT EVERY ROW control block. In this case, you must pass all the values for each row of the report.
- When you use the AFTER GROUP OF control block. In this case, you must pass at least the parameters named in the AFTER GROUP OF statement.

The DEFINE section obeys the same rules as given in Chapter 7 for the DEFINE statement, except that report parameters may not be of type ARRAY nor records with ARRAY members.

OUTPUT Section

An INFORMIX-4GL REPORT routine 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 INFORMIX-4GL report to a file or the system printer. On UNIX systems, you can also direct output to a pipe.

The OUTPUT section consists of the OUTPUT keyword, followed by one or more statements. The structure of an OUTPUT section is shown below.

You cannot use more than one of the REPORT TO options in a REPORT routine. When you do not use one of the REPORT TO options, INFORMIX-4GL sends the report to your screen.

```
OUTPUT
   [REPORT TO statement]
   [LEFT MARGIN statement]
   [RIGHT MARGIN statement]
   [TOP MARGIN statement]
   [BOTTOM MARGIN statement]
   [PAGE LENGTH statement]
```

REPORT TO

Overview

This statement directs the output of the **INFORMIX-4GL** report to a file, to an operating system pipe (UNIX systems only), or to the system printer.

Syntax

REPORT TO {"*filename*" | PIPE *program* | PRINTER}

Explanation

REPORT TO	are required keywords.
filename	is a quoted string containing the name of a system file that will receive the report.
PIPE	is an optional keyword.
program	is a variable of type CHAR or a quoted string containing the name of a UNIX program that is to receive the output from the **INFORMIX-4GL** report. The program name, and any associated arguments, must be within quotation marks.
PRINTER	is an optional keyword.

Notes

1. If the START REPORT statement has a TO clause directing the output of the report to a file, pipe, or printer, **INFORMIX-4GL** ignores the REPORT TO statement of the OUTPUT section.

2. If *filename* is a variable, you must pass it as an argument to the REPORT routine.

3. The REPORT TO PRINTER statement causes INFORMIX-4GL to send the report to the program named by the DBPRINT environment variable. If you do not set this variable, INFORMIX-4GL sends the report to the **lp** program (on UNIX systems) or the **lpt1** printer (on DOS systems).

4. 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. On UNIX systems, you can also use the REPORT TO PIPE statement to direct the output to a program that sends the output to the correct printer.

Examples

The following OUTPUT section directs the report output to the **label.out** system file.

```
OUTPUT
    REPORT TO "label.out"
    LEFT MARGIN 0
    TOP MARGIN 0
    BOTTOM MARGIN 0
    PAGE LENGTH 6
```

The following OUTPUT section directs the output from the INFORMIX-4GL report to the **more** utility.

```
OUTPUT
    REPORT TO PIPE "more"
```

LEFT MARGIN

Overview

This statement sets a left margin for a report.

Syntax

```
LEFT MARGIN integer
```

Explanation

LEFT MARGIN	are required keywords.
integer	is an integer that specifies the width of the left margin, in spaces.

Notes

1. The default left margin is five spaces.

2. All columnar displacement indicated by the COLUMN function starts at the margin set by LEFT MARGIN.

Examples

The following LEFT MARGIN statement instructs INFORMIX-4GL to print the left side of the report as far to the left as possible.

```
OUTPUT
    REPORT TO "label.out"
    LEFT MARGIN 0
    TOP MARGIN 0
    BOTTOM MARGIN 0
    PAGE LENGTH 6
```

RIGHT MARGIN

Overview

This statement sets a right margin for a report.

Syntax

RIGHT MARGIN *integer*

Explanation

RIGHT MARGIN	are required keywords.
integer	is an integer that specifies the width of the text on the page, in characters.

Notes

1. The RIGHT MARGIN determines the right margin by specifying the width of the page, in characters. It is not dependent on the LEFT MARGIN, but always starts its count from the left edge of the page.

2. The RIGHT MARGIN is effective only when the FORMAT section contains an EVERY ROW statement.

3. The default RIGHT MARGIN is 132.

4. INFORMIX-4GL attempts to produce an EVERY ROW report by listing the variable names across the top of the page and presenting the data in columns beneath these headings. If there is not sufficient room between the LEFT MARGIN and the RIGHT MARGIN to do this, INFORMIX-4GL produces a report that lists the variable names and the data of each row in two columns.

Examples

The following example demonstrates the use of the RIGHT MARGIN statement. INFORMIX-4GL sets the right margin for the report at 70 characters.

```
REPORT simple(customer)
DEFINE customer LIKE customer.*
OUTPUT
   RIGHT MARGIN 70

FORMAT
   EVERY ROW
END REPORT
```

TOP MARGIN

Overview

This statement sets a top margin for a report.

Syntax

```
TOP MARGIN integer
```

Explanation

TOP MARGIN	are required keywords.
integer	is an integer that specifies the number of blank lines that INFORMIX-4GL leaves at the top of each page.

Notes

1. The default top margin is three lines.
2. The top margin appears above any page header you specify.

Examples

The following TOP MARGIN statement instructs INFORMIX-4GL to begin printing at the top of each page.

```
OUTPUT
    REPORT TO "label.out"
    LEFT MARGIN 0
    TOP MARGIN 0
    BOTTOM MARGIN 0
    PAGE LENGTH 6
```

BOTTOM MARGIN

Overview

This statement sets a bottom margin for a report.

Syntax

```
BOTTOM MARGIN integer
```

Explanation

BOTTOM MARGIN	are required keywords.
integer	is an integer that specifies the number of blank lines that **INFORMIX-4GL** is to leave at the bottom of each page.

Notes

1. The default bottom margin is three lines.
2. The bottom margin appears below any page trailer you specify.

Examples

The following BOTTOM MARGIN statement instructs INFORMIX-4GL to continue printing to the bottom of each page.

```
OUTPUT
    REPORT TO "label.out"
    LEFT MARGIN 0
    TOP MARGIN 0
    BOTTOM MARGIN 0
    PAGE LENGTH 6
```

PAGE LENGTH

Overview

This statement sets the number of lines on each page of a report.

Syntax

PAGE LENGTH *integer*

Explanation

PAGE LENGTH	are required keywords.
integer	is an integer that specifies the length of the page, in lines.

Notes

1. The default page length is sixty-six lines.

2. The PAGE LENGTH includes the TOP MARGIN and BOTTOM MARGIN.

Examples

The following example demonstrates the use of the PAGE LENGTH statement. INFORMIX-4GL prints each page with 22 lines. On a standard 24-line video screen, a page length of 22 lines is the longest you can use with the PAUSE statement without causing undesirable scrolling.

```
OUTPUT
    PAGE LENGTH 22
    TOP MARGIN 0
    BOTTOM MARGIN 0
```

ORDER BY Section

The optional ORDER BY section specifies the variables on which you want the rows to be sorted and the order in which INFORMIX-4GL will process group control blocks in the FORMAT section. You should have an ORDER BY section if you have used group control blocks in your report and

- you have not sorted the input rows, or
- you have already sorted the input rows and you want to specify the exact order in which the group control blocks are processed. (Without the ORDER BY section INFORMIX-4GL chooses the order in which to process the group control blocks.) In this case, use the EXTERNAL keyword so that the rows will not be sorted again.

ORDER [EXTERNAL] BY *sort-list*

Explanation

ORDER BY	are required keywords.
EXTERNAL	is an optional keyword.
sort-list	is a list of one or more variables from the list of arguments to the REPORT routine.

Notes

1. The ORDER BY section specifies two things. First, it specifies the order in which INFORMIX-4GL orders the input rows. If *sort-list* contains **a**, **b**, and **c** in that order, then INFORMIX-4GL orders the input rows first by **a**. Within that ordering, INFORMIX-4GL orders the rows next by **b**. Finally, INFORMIX-4GL orders the resulting rows by **c**.

Second, the ORDER BY section specifies the order in which INFORMIX-4GL will process group control blocks. (See the section "Control Blocks" for more information.)

2. The EXTERNAL keyword in the ORDER BY section specifies that the input rows are already sorted. INFORMIX-4GL does not resort the rows in this case and prints the report with one pass through the input rows.

3. If there is an ORDER BY section without the EXTERNAL keyword, INFORMIX-4GL makes two passes through the input data. During the first pass, it sorts the data and stores it in a temporary file. During the second pass, it prints the report. Since sorting within a SELECT statement makes use of indexes while sorting within the REPORT routine does not, you may find that external sorting increases the performance of your program. You may also want to sort your data outside of the REPORT routine if your program experiences memory limitations.

   If the input rows for your report come from the rows returned by only one cursor, you should use the ORDER BY clause in the SELECT statement associated with the cursor, and use the EXTERNAL keyword in the ORDER BY section of your report.

4. If you have just one variable named in group control blocks and the input rows are already sorted, then you do not need an ORDER BY section.

FORMAT Section

An INFORMIX-4GL REPORT routine must contain a FORMAT section. The FORMAT section determines what the report will look like. It works with the data that is passed to the routine through the argument list or with data that you put in global variables for each row of the report.

The FORMAT section begins with the FORMAT keyword and ends at the end of the REPORT routine, that is, with the END REPORT keywords.

There are two major types of FORMAT sections. The simplest FORMAT section contains only an EVERY ROW statement between the FORMAT and END REPORT 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
END REPORT
```

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 FORMAT statement such as PRINT or SKIP *n* LINES and may contain other INFORMIX-4GL statements.

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. The following shows the structure of this type of FORMAT section.

```
FORMAT
     [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
     ...]
END REPORT
```

EVERY ROW

Overview

The EVERY ROW statement causes INFORMIX-4GL to output every row that you pass to the report. It uses a default format.

Syntax

```
EVERY ROW
```

Explanation

EVERY ROW	are required keywords.

Notes

1. The report consists of only the data that you pass to the routine through its arguments.

2. This statement is useful when you want to run a quick report using a default format.

3. The EVERY ROW statement stands by itself—you cannot modify it with any of the statements listed in the "Statements" section of this chapter.

4. When you use the EVERY ROW statement, you cannot use any control blocks in the FORMAT section.

5. A report generated by an EVERY ROW statement uses the variable names that you pass as arguments to the routine at run time as column headings.

6. If the variables you passed as arguments fit on a line, INFORMIX-4GL will produce a report with variable names across the top of each page; otherwise, it will produce a report with the variable names down the left side of the page.

7. You can use the RIGHT MARGIN statement in the OUTPUT section to control the width of a report that uses the EVERY ROW statement.

8. 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 in the "Control Blocks" section later in this chapter.)

Examples

The following example shows a minimal INFORMIX-4GL REPORT routine using the EVERY ROW statement.

```
REPORT minimal(customer)
DEFINE customer RECORD LIKE customer.*
FORMAT
   EVERY ROW
END REPORT
```

The following display shows a portion of the output from the preceding specification.

```
customer.customer_num  101
customer.fname         Ludwig
customer.lname         Pauli
customer.company       All Sports Supplies
customer.address1      213 Erstwild Court
customer.address2
customer.city          Sunnyvale
customer.state         CA
customer.zipcode       94086
customer.phone         408-789-8075

customer.customer_num  102
customer.fname         Carole
customer.lname         Sadler
customer.company       Sports Spot
customer.address1      785 Geary St
customer.address2
customer.city          San Francisco
customer.state         CA
customer.zipcode       94117
customer.phone         415-822-1289

customer.customer_num  103
customer.fname         Philip
customer.lname         Currie
      .
      .
      .
```

Below is another example of a brief report specification that uses the EVERY ROW statement.

```
REPORT simple(order_num, customer_num, order_date)
DEFINE order_num LIKE orders.order_num,
       customer_num LIKE orders.customer_num,
       order_date LIKE orders.order_date
FORMAT
   EVERY ROW
END REPORT
```

The following display shows the output from the preceding REPORT routine.

```
order_num customer_num order_date

     1001          104 01/20/1986
     1002          101 06/01/1986
     1003          104 10/12/1986
     1004          106 04/12/1986
     1005          116 12/04/1986
     1006          112 09/19/1986
     1007          117 03/25/1986
     1008          110 11/17/1986
     1009          111 02/14/1986
     1010          115 05/29/1986
     1011          104 03/23/1986
     1012          117 06/05/1986
     1013          104 09/01/1986
     1014          106 05/01/1986
     1015          110 07/10/1986
```

Control Blocks

Control blocks provide the structure for a custom report. 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 routine.

Each control block must include at least one statement (see the "Statements" section later in this chapter).

When you use the BEFORE GROUP OF, AFTER GROUP OF, and ON EVERY ROW control blocks in a single REPORT routine, INFORMIX-4GL processes all BEFORE GROUP OF blocks before the ON EVERY ROW block and the ON EVERY ROW block before all AFTER GROUP OF blocks. The order in which INFORMIX-4GL processes the BEFORE GROUP OF control blocks and AFTER GROUP OF control blocks depends upon the hierarchy of variables listed in the ORDER BY section or, in the absence of an ORDER BY section, implied by the order of first mention of variables in either BEFORE or AFTER GROUP OF control blocks. Assume that the ORDER BY section orders by variables **a**, **b**, and **c**. Then the following display indicates the order by which INFORMIX-4GL processes control blocks:

```
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
```

Figure 4-1. Order of Group Processing

AFTER GROUP OF

Overview

The AFTER GROUP OF control block specifies the action INFORMIX-4GL takes after it processes a group of rows. Grouping is determined by the ordering you did earlier.

Syntax

```
AFTER GROUP OF variable-name
  statement
  ...
```

Explanation

AFTER GROUP OF	are required keywords.
variable-name	is the name of one of the variables passed as an argument.
statement	is a FORMAT or INFORMIX-4GL statement.

Notes

1. You must pass at least the value of *variable-name* through the arguments of the REPORT routine.

2. A *group* of rows is all rows that contain the same value for a given variable. INFORMIX-4GL automatically groups rows when you use an ORDER BY section of a REPORT routine or the ORDER BY clause in a SELECT statement—that is, groups come together when you order a list.

When you specify more than one column in the ORDER BY section or clause, INFORMIX-4GL orders the rows first by the first variable you specify (most significant), second by the second variable you specify, and so on, until the last variable you specify (least significant) is ordered.

INFORMIX-4GL 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 (refer to Figure 4-1 at the beginning of the "Control Blocks" section).

3. You can have one AFTER GROUP OF control block for each variable on which you have ordered the data.

4. If you have an ORDER BY section and you have more than one AFTER GROUP OF control block, their order within the FORMAT section is not significant.

5. When INFORMIX-4GL 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.

6. Group aggregates can be used only in AFTER GROUP OF control blocks. You cannot use group aggregates in any other control blocks.

7. When INFORMIX-4GL processes the statements in an AFTER GROUP OF control block, the variables of the report 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.

Examples

```
AFTER GROUP OF r.order_num
   PRINT " ",r.order_date,7 SPACES,r.order_num USING "###&",
  8 SPACES,r.ship_date,"  ",
  GROUP SUM(r.total_price) USING "$$$$,$$$,$$$.&&"

AFTER GROUP OF r.customer_num
   PRINT 42 SPACES,"---------------"
   PRINT 42 SPACES,GROUP SUM(r.total_price) USING "$$$$,$$$,$$$.&&"
```

BEFORE GROUP OF

Overview

The BEFORE GROUP OF control block specifies what action INFORMIX-4GL takes before it processes a group of rows. Grouping is determined by the ordering you did earlier.

Syntax

```
BEFORE GROUP OF variable-name
  statement
  ...
```

Explanation

BEFORE GROUP OF	are required keywords.
variable-name	is the name of one of the variables passed as an argument.
statement	is a FORMAT or INFORMIX-4GL statement.

Notes

1. You must pass at least the value of *variable-name* through the arguments of the REPORT routine.

2. A *group* of rows is all rows that contain the same value for a given variable. INFORMIX-4GL automatically groups rows when you use an ORDER BY section of a REPORT routine or the ORDER clause of a SELECT statement—that is, groups come together when you order a list.

When you specify more than one variable in an ORDER BY section or clause, INFORMIX-4GL orders the rows first by the first variable you specify (most significant), second by the second variable you specify, and so on, until the last variable you specify (least significant) is ordered.

INFORMIX-4GL processes the statements in a BEFORE GROUP OF control block at the start of a report, each time the specified variable changes value, and each time a more significant variable changes value (refer to Figure 4-1 at the beginning of the "Control Blocks" section).

3. You can have one BEFORE GROUP OF control block for each variable that you order.

4. If you have an ORDER BY section and you have more than one BEFORE GROUP OF control block, their order within the FORMAT section is not significant.

5. When INFORMIX-4GL starts to generate a report, it executes all the statements in the BEFORE GROUP OF control blocks before it executes those in the ON EVERY ROW control block.

6. You can use a SKIP TO TOP OF PAGE statement in a BEFORE GROUP OF control block to cause each group to start at the top of a page.

7. When INFORMIX-4GL processes the statements in a BEFORE GROUP OF control block, the report variables have the values from the first row of the new group. From this perspective, the BEFORE GROUP OF control block could be called the "on first row of group" control block.

Examples

```
BEFORE GROUP OF r.customer_num
   SKIP TO TOP OF PAGE
```

FIRST PAGE HEADER

Overview

The FIRST PAGE HEADER control block specifies what information appears at the top of the first page of the report.

Syntax

```
FIRST PAGE HEADER
        statement
        ...
```

Explanation

FIRST PAGE HEADER	are required keywords.
statement	is an INFORMIX-4GL or FORMAT statement.

Notes

1. The TOP MARGIN (set in the OUTPUT section) affects how close to the top of the page INFORMIX-4GL displays the page header.
2. A FIRST PAGE HEADER control block overrides a PAGE HEADER control block on the first page of a report.
3. You cannot use the SKIP TO TOP OF PAGE statement in a FIRST PAGE HEADER control block.
4. If you use an IF THEN ELSE statement in a FIRST PAGE HEADER control block, the number of lines displayed by the PRINT statements following the THEN keyword must be equal to the number of lines displayed by the PRINT statements following the ELSE keyword.

5. You cannot use the PRINT *filename* statement to read and display text from a file in a FIRST PAGE HEADER control block.

6. You can use a FIRST PAGE HEADER control block to produce a title page, as well as column headings.

Examples

This example is from a report that produces multiple labels across the page.

```
FIRST PAGE HEADER
   {Nothing is displayed in this
    control block.  It just
    initializes variables that are
    used in the ON EVERY ROW
    control block.}

   {Initialize label counter.}
   LET i = 1

   {Determine label width (allow
    eight spaces total between labels).}
   LET l_size = 72/count1

   {Divide the eight spaces between
    the number of labels across the page.}
   LET white = 8/count1
```

This FIRST PAGE HEADER does not display any information. Because INFORMIX-4GL executes the FIRST PAGE HEADER control block *before* it generates any output, you can use this control block to initialize variables that you use in the FORMAT section.

ON EVERY ROW

Overview

The ON EVERY ROW control block specifies what action INFORMIX-4GL is to take for every row of data that you pass to the routine.

Syntax

```
ON EVERY ROW
        statement
        ...
```

Explanation

ON EVERY ROW	are required keywords.
statement	is an INFORMIX-4GL or FORMAT statement.

Notes

1. INFORMIX-4GL 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 appropriate BEFORE GROUP OF control blocks will be 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 appropriate AFTER GROUP OF control blocks will be executed (in the reverse order of their significance) after the ON EVERY ROW control block is executed.

Examples

The following example is from a report that lists all the customers, their addresses, and their telephone numbers across the page.

```
ON EVERY ROW
   PRINT customer_num USING "####",
        COLUMN 12, fname CLIPPED, 1 SPACE,
        lname CLIPPED, COLUMN 35, city CLIPPED,
        ", " , state,  COLUMN 57, zipcode,
        COLUMN 65, phone
```

The following example is from a mailing label report.

```
ON EVERY ROW
   IF (city IS NOT NULL) AND (state IS NOT NULL) THEN
        PRINT fname CLIPPED, 1 SPACE, lname
        PRINT company
        PRINT address1
        IF (address2 IS NOT NULL) THEN
           PRINT address2
        PRINT city CLIPPPED ", " , state, 2 SPACES, zipcode
        SKIP TO TOP OF PAGE
   END IF
```

ON LAST ROW

Overview

The ON LAST ROW control block specifies what action INFORMIX-4GL is to take after it processes the last row passed to the REPORT routine.

Syntax

```
ON LAST ROW
        statement
        ...
```

Explanation

ON LAST ROW	are required keywords.
statement	is an INFORMIX-4GL or FORMAT statement.

Notes

1. INFORMIX-4GL 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.

2. You can use the ON LAST ROW control block to display report totals.

3. When INFORMIX-4GL processes the statements in an ON LAST ROW control block, the columns that the report is processing still have the values from the final row that the report processed.

Examples

```
ON LAST ROW
 SKIP 1 LINE
 PRINT COLUMN 12, "TOTAL NUMBER OF CUSTOMERS:",
       COLUMN 57, COUNT(*) USING "##"
```

PAGE HEADER

Overview

The PAGE HEADER control block specifies what information appears at the top of each page of the report.

Syntax

```
PAGE HEADER
        statement
        ...
```

Explanation

PAGE HEADER	are required keywords.
statement	is an INFORMIX-4GL or FORMAT statement.

Notes

1. The TOP MARGIN (in the OUTPUT section) affects how close to the top of the page INFORMIX-4GL displays the page header.

2. A FIRST PAGE HEADER control block overrides a PAGE HEADER control block on the first page of a report.

3. You cannot use the SKIP TO TOP OF PAGE statement in a PAGE HEADER control block.

4. The number of lines produced by the PAGE HEADER control block may not change from page to page and must be unambiguously expressed. The following rules are special cases of this more general principle:

- You cannot have a SKIP *integer* LINES inside a loop in the PAGE HEADER control block.

- You cannot use a NEED statement in the 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 statements following the THEN keyword must be equal to the number of lines displayed by the PRINT statements following the ELSE keyword.

- If you use a CASE, 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 will suppress RETURNs in the loop, keeping the number of lines in the header constant from page to page.

- You cannot use a PRINT *filename* statement to read and display text from a file in a PAGE HEADER control block.

5. You can use a PAGE HEADER control block to display column headings in a report.

6. 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.

Examples

The following example produces the column headings for printing the customer data across the page.

```
PAGE HEADER
  PRINT "NUMBER",
  COLUMN 12, "NAME",
  COLUMN 35, "LOCATION",
  COLUMN 57, "ZIP",
  COLUMN 65, "PHONE"
  SKIP 1 LINE
```

PAGE TRAILER

Overview

The PAGE TRAILER control block specifies what information appears at the bottom of each page of the report.

Syntax

```
PAGE TRAILER
        statement
        ...
```

Explanation

PAGE TRAILER	are required keywords.
statement	is an INFORMIX-4GL or FORMAT statement.

Notes

1. The BOTTOM MARGIN (in the OUTPUT section) affects how close to the bottom of the page INFORMIX-4GL displays the page trailer.

2. You cannot use the SKIP TO TOP OF PAGE statement in a PAGE TRAILER control block.

3. The number of lines produced by the PAGE TRAILER control block may not change from page to page and must be unambiguously expressed. The following rules are special cases of this more general principle:

   - You cannot have a SKIP *integer* LINES inside a loop in the PAGE TRAILER control block.

- You cannot use a NEED statement in the 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 statements following the THEN keyword must be equal to the number of lines displayed by the PRINT statements following the ELSE keyword.

- If you use a CASE, 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 will suppress RETURNs in the loop, keeping the number of lines in the header constant from page to page.

- You cannot use a PRINT *filename* statement to read and display text from a file in a PAGE TRAILER control block.

4. INFORMIX-4GL executes the PAGE TRAILER control block before the PAGE HEADER control block when you issue a SKIP TO TOP OF PAGE statement anywhere.

5. 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.

Examples

```
PAGE TRAILER
    PRINT COLUMN 28,
        PAGENO USING "page <<<<"
```

Statements

The control blocks determine *when* INFORMIX-4GL takes an action in a report, while the statements determine *what* action INFORMIX-4GL takes. You may use any INFORMIX-4GL statement in a control block, as well as a number of statements that can be used only in the FORMAT section of a REPORT routine. The most common INFORMIX-4GL statements used in reports are FOR, IF, LET, and WHILE. The FORMAT statements are as follows:

NEED	PRINT FILE
PAUSE	SKIP
PRINT	

A description of these FORMAT-only statements follows.

NEED

Overview

This statement causes subsequent display to start on the next page if there is not the specified number of lines remaining on the current page.

Syntax

NEED *num-expr* LINES

Explanation

NEED	is a required keyword.
num-expr	is an expression that evaluates to an integer specifying the number of lines needed.
LINES	is a required keyword.

Notes

1. Use the NEED statement to prevent INFORMIX-4GL from splitting parts of the report that you want to keep together on a single page.

2. INFORMIX-4GL does not include the BOTTOM MARGIN value in the number of lines counted.

3. If INFORMIX-4GL triggers the NEED statement in printing a report, it prints both the PAGE TRAILER and the PAGE HEADER.

4. You may not use this statement in PAGE HEADER or PAGE TRAILER control blocks.

PAUSE

Overview

This statement causes output to the terminal to pause until you press **RETURN**.

Syntax

PAUSE [*string*]

Explanation

PAUSE	is a required keyword.
string	is an optional message that PAUSE displays. If you do not supply a message, PAUSE will not display a message.

Notes

1. If you use a REPORT TO *filename* or REPORT TO PRINTER statement in the OUTPUT section or in the START REPORT statement, the PAUSE statement will have no effect. (It also will not work with the REPORT TO PIPE statement available on UNIX systems.)

Examples

The following example will cause INFORMIX-4GL to pause while running the report.

```
AFTER GROUP OF item_num
       .
       .
       .
SKIP TO TOP OF PAGE
PAUSE "Press RETURN to continue"
```

PRINT

Overview

This statement displays information as specified in the OUTPUT section.

Syntax

PRINT [*exprlist*][;]

Explanation

PRINT	is a required keyword.
exprlist	is an optional list of one or more expressions, separated by commas.
;	is an optional keyword that suppresses a RETURN at the end of the line.

Notes

1. One PRINT statement displays its output on one line, no matter how many lines the statement occupies in the report specification.

2. Unless you use the keyword CLIPPED or USING following an expression, INFORMIX-4GL displays variables with a width depending on their data type, as shown in Figure 4-2.

Data Type	Default Size
CHAR	declared size
DATE	10
FLOAT	14 (including sign and decimal point)
SMALLINT	6 (including sign)
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)

Figure 4-2. Default Display Widths

Examples

The following example is from a mailing label report:

```
FORMAT
  ON EVERY ROW
     PRINT fname, lname
     PRINT company
     PRINT address1
     PRINT address2
     PRINT city, ", " , state, 2 SPACES, zipcode
     SKIP 2 LINES
```

The following example is from a report that prints the customer list.

```
FIRST PAGE HEADER
  PRINT COLUMN 30, "CUSTOMER LIST"
  SKIP 2 LINES
  PRINT "Listings for the State of ", thisstate
  SKIP 2 LINES
  PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35, "LOCATION",
    COLUMN 57, "ZIP", COLUMN 65, "PHONE"
  SKIP 1 LINE

PAGE HEADER
  PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35, "LOCATION",
    COLUMN 57, "ZIP", COLUMN 65, "PHONE"
  SKIP 1 LINE

ON EVERY ROW
 PRINT customer_num USING "####",
 COLUMN 12, fname CLIPPED, 1 SPACE,
   lname CLIPPED, COLUMN 35, city CLIPPED, ", " , state,
   COLUMN 57, zipcode, COLUMN 65, phone
```

PRINT FILE

Overview

This statement displays the contents of a text file in a report.

Syntax

```
PRINT FILE "filename"
```

Explanation

PRINT FILE	are required keywords.
filename	is a required filename that can be a pathname. You must enclose the filename in quotation marks.

Notes

1. You can use the PRINT FILE statement to include the body of a form letter in a report that generates custom letters.

SKIP

Overview

This statement skips lines in a report or skips to the top of the next page.

Syntax

```
SKIP {integer LINE[S] | TO TOP OF PAGE}
```

Explanation

SKIP	is a required keyword.
integer	is an integer specifying the number of lines to skip.
LINES	is an optional keyword. You can use the keyword LINE in place of LINES if you like.
TO TOP OF PAGE	are optional keywords.

Notes

1. You cannot use a SKIP LINES statement inside a CASE, FOR, or WHILE statement.

2. You cannot use a SKIP TO TOP OF PAGE statement in a FIRST PAGE HEADER, PAGE HEADER, or PAGE TRAILER control block.

Examples

The following example is from a report that prints the customer list.

```
FIRST PAGE HEADER
  PRINT COLUMN 30, "CUSTOMER LIST"
  SKIP 2 LINES
  PRINT "Listings for the State of ", thisstate
  SKIP 2 LINES
  PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35, "LOCATION",
    COLUMN 57, "ZIP", COLUMN 65, "PHONE"
  SKIP 1 LINE

PAGE HEADER
  PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35, "LOCATION",
    COLUMN 57, "ZIP", COLUMN 65, "PHONE"
  SKIP 1 LINE

ON EVERY ROW
 PRINT customer_num USING "####",
 COLUMN 12, fname CLIPPED, 1 SPACE,
   lname CLIPPED, COLUMN 35, city CLIPPED, ", " , state,
   COLUMN 57, zipcode, COLUMN 65, phone
```

The following example is from a mailing label report.

```
FORMAT
ON EVERY ROW
  IF (city IS NOT NULL) AND (state IS NOT NULL) THEN
     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 IF
```

Expressions and Built-in Functions

Expressions used within REPORT routines have the same syntax as expressions used elsewhere in INFORMIX-4GL. These expressions may use the functions defined in Chapter 1. You may also use the built-in aggregate functions. The aggregates in reports have a slightly different application from those used in a SELECT statement. These differences are described in the next pages. There are also built-in functions that you may use only in REPORT routines. The table below lists all the functions you may use in a REPORT routine.

ASCII[a]	MDY()[a]
AVG[rs]	MONTH()[a]
CLIPPED[a]	PAGENO[r]
COLUMN[a]	PERCENT(*)[r]
COUNT(*)[rs]	SPACES[a]
DATE[a]	SUM[rs]
DATE()[a]	TIME[a]
DAY()[a]	TODAY[a]
GROUP[r]	USING[a]
LINENO[r]	WEEKDAY()[a]
MAX[rs]	YEAR()[a]
MIN[rs]	

r You can use these functions only within the FORMAT section of a REPORT routine. A description of these functions follows.

rs You can use these functions only within the FORMAT section of a REPORT routine or in INSERT, SELECT, or UPDATE statements elsewhere. They are described both in the following pages and in Chapter 7.

a These functions are described in Chapter 1.

Aggregates

Overview

Aggregates allow you to summarize information in a report.

Syntax

```
[GROUP]
    {COUNT(*) | PERCENT(*) | {SUM | AVG | MIN | MAX}(expr1)}
    [WHERE expr2]
```

Explanation

GROUP	is an optional keyword that causes the aggregate to reflect information for a specific group only. You can only use this keyword in an AFTER GROUP OF control block.
COUNT(*)	is a keyword. This keyword is always evaluated as the total number of rows qualified by the optional WHERE clause.
PERCENT(*)	is the keyword that evaluates COUNT as a percent of the total number of rows in the report.
SUM	evaluates as the total of *expr1* in the rows qualified by the optional WHERE clause. SUM ignores rows with NULL value for *expr1*; it returns NULL if all rows have a NULL value for *expr1*.
AVG	evaluates as the average of *expr1* in the rows qualified by the optional WHERE clause. AVG ignores rows with NULL value for *expr1*; it returns NULL if all rows have a NULL value for *expr1*.

MIN	evaluates as the minimum of *expr1* in the rows qualified by the optional WHERE clause. MIN ignores rows with NULL value for *expr1*; it returns NULL if all rows have a NULL value for *expr1*.
MAX	evaluates as the maximum of *expr1* in the rows qualified by the optional WHERE clause. MAX ignores rows with NULL value for *expr1*; it returns NULL if all rows have a NULL value for *expr1*.
expr1	is the expression that SUM, AVG, MIN, and MAX evaluate. It is typically a numeric variable or a numeric expression that includes a numeric variable.
WHERE	is an optional keyword.
expr2	is a Boolean expression that qualifies the aggregate.

Notes

1. The WHERE clause allows you to select among the rows passed to the report.

Examples

```
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 "$$$$.&&"
  SKIP 1 LINE
  PRINT 5 SPACES, "Total amount for the order:  ",
    ship_charge + GROUP SUM(total_price) USING "$$,$$$,$$$.&&"
```

LINENO

Overview

This expression has the value of the line number of the report line that INFORMIX-4GL is currently printing. INFORMIX-4GL computes the line number by calculating the number of lines from the top of the page including the TOP MARGIN.

Syntax

LINENO

Explanation

LINENO	is a required keyword.

Examples

```
    PRINT COLUMN 10, LINENO USING "Line <<<"
```

PAGENO

Overview

This expression has the value of the page number of the page that INFORMIX-4GL is currently printing.

Syntax

```
PAGENO
```

Explanation

PAGENO	is a required keyword.

Notes

1. 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.)

Examples

```
PAGE TRAILER
   PRINT COLUMN 28, PAGENO USING "page <<<<"
```

TIME

Overview

This expression evaluates as a character string with a value of the current time in the format *hh:mm:ss*.

Syntax

```
TIME
```

Explanation

TIME	is a required keyword.

Chapter 5

4GL Function Library

Chapter 5 Table of Contents

Chapter Overview 5

- ARG_VAL 6
- ARR_COUNT 8
- ARR_CURR 10
- DOWNSHIFT 12
- ERR_GET 13
- ERR_PRINT 14
- ERR_QUIT 15
- ERRORLOG 16
- INFIELD 18
- LENGTH 20
- NUM_ARGS 21
- SCR_LINE 23
- SET_COUNT 25
- SHOWHELP 26
- STARTLOG 28
- UPSHIFT 30

Chapter Overview

This chapter describes functions that are available from the INFORMIX-4GL library. You can call any of the functions listed in the following table and the INFORMIX-4GL compiler automatically links them in the final program.

Function	**Return Value or Effect**
arg_val	Command line argument
arr_count	Total filled rows of program array
arr_curr	Current row of program array
downshift	Lowers case of string argument
err_print	Prints 4GL error message on ERROR line
err_get	Prints 4GL error message to string variable
err_quit	Prints 4GL error message and exits
errorlog	Appends argument to error log
infield	True if argument is current field
length	Length in bytes of string argument
num_args	Number of command line arguments
scr_line	Current row of screen array
set_count	Sets number of rows of program array
showhelp	Displays help message
startlog	Opens error log file
upshift	Raises case of string argument

These functions are described more fully in the following pages.

ARG_VAL

Overview

arg_val returns a command line argument as a CHAR variable.

Syntax

```
arg_val(expr)
```

Explanation

expr	is an integer expression.

Notes

1. **arg_val**(*n*) returns the n^{th} command line argument as a CHAR variable.

2. The value of *expr* must be between 0 and **num_args ()**, the number of command line arguments. **arg_val(0)** is your program name.

3. **arg_val** and **num_args** allow you to pass data to the program from the command line that executes the program.

Examples

If your compiled executable 4GL program name is **myprog** and you executed the command line

```
myprog john mary ellen joe frank
```

the following program segment puts the names you entered into an array of CHAR variables.

```
DEFINE args ARRAY[8] OF CHAR(10),
       i    SMALLINT

FOR i = 1 TO num_args()
    LET args[i] = arg_val(i)
END FOR
```

Related Functions

NUM_ARGS

ARR_COUNT

Overview

arr_count returns the number of rows that are entered in a program array during or after an INPUT ARRAY statement.

Syntax

```
arr_count( )
```

Notes

1. You can use **arr_count** to record the number of rows currently stored in a program array.

2. If you execute a statement in a BEFORE, AFTER, or ON KEY clause that changes the value of **arr_curr**, **scr_line**, or **arr_count**, the new value remains for the duration of the clause (unless changed again.) After INFORMIX-4GL executes the last statement in the clause, INFORMIX-4GL restores the original value.

Examples

```
FUNCTION insert_items()
   DEFINE counter SMALLINT
   FOR counter = 1 TO arr_count()
      INSERT INTO items
         VALUES (p_items[counter].item_num,
            p_orders.order_num,
            p_items[counter].stock_num,
            p_items[counter].manu_code,
            p_items[counter].quantity,
            p_items[counter].total_price)
   END FOR
END FUNCTION
```

Related Functions

ARR_CURR, SCR_LINE

ARR_CURR

Overview

arr_curr returns the row of the program array that corresponds to the current screen array row during or immediately after an INPUT ARRAY or DISPLAY ARRAY statement.

Syntax

```
arr_curr( )
```

Notes

1. The current screen row is the row where the cursor is located at the beginning of a BEFORE ROW or AFTER ROW clause.
2. The first row of both the program array and the screen array is numbered 1.
3. If you execute a statement in a BEFORE, AFTER, or ON KEY clause that changes the value of **arr_curr**, **scr_line**, or **arr_count**, the new value remains for the duration of the clause (unless changed again.) After **INFORMIX-4GL** executes the last statement in the clause, **INFORMIX-4GL** restores the original value.

For example, if the DISPLAY ARRAY statement in the following INPUT ARRAY statement changes the value of **arr_curr** from 1 to 4, the value remains 4 for the rest of the ON KEY clause. After **INFORMIX-4GL** executes the MESSAGE statement, it resets **arr_curr** to 1, its prior value.

```
INPUT ARRAY p_items to s_items.*
   ON KEY (F6)
   DISPLAY ARRAY p_stock to s_stock.*
   END DISPLAY
   LET pa_curr = arr_curr()
   MESSAGE "The current value of arr_curr is: ", pa_curr
END INPUT
```

Examples

The following example tests the user input and rejects it if the customer is not from California. (See the definition of **scr_line()**.)

```
DEFINE p_array ARRAY[90] OF RECORD
     fname    CHAR(15),
     lname    CHAR(15),
     state    CHAR(2)
     END RECORD,
     pa_curr,
     sc_curr SMALLINT

INPUT ARRAY p_array FROM scr_array.*
     AFTER FIELD state
          LET pa_curr = arr_curr()
          LET sc_curr = scr_line()
          IF upshift(p_array[pa_curr].state) != "CA" THEN
               ERROR "Customers must be from California"
               INITIALIZE p_array[pa_curr].* TO NULL
               CLEAR scr_array[sc_curr].*
               NEXT FIELD fname
          END IF
END INPUT
```

Related Functions

ARR_COUNT, SCR_LINE

DOWNSHIFT

Overview

downshift returns a string in which all uppercase characters in its argument are converted to lowercase.

Syntax

downshift(*str*)

Explanation

str	is a string constant or a variable of type CHAR.

Notes

1. Non-alphabetic characters in *str* are not altered.
2. You can use the **downshift** function in an expression (when such usage is allowed) or assign the value returned by the function to a variable.

Examples

```
LET d_str = downshift(str)

DISPLAY d_str
```

Related Functions

UPSHIFT

ERR_GET

Overview

err_get returns a CHAR variable which contains the INFORMIX-4GL error message corresponding to its argument.

Syntax

err_get(*expr*)

Explanation

expr	is an integer expression.

Notes

1. *expr* is usually the global **status** variable.
2. **err_get** is most useful during debugging. The message it returns is probably not helpful to the user of your application.

Examples

```
. . .
IF status < 0 THEN
    LET errtext = err_get(status)
END IF
```

Related Functions

ERR_PRINT, ERR_QUIT, STARTLOG

ERR_PRINT

Overview

err_print prints the INFORMIX-4GL error message corresponding to its argument on the Error Line.

Syntax

```
call err_print(expr)
```

Explanation

expr	is an integer expression.

Notes

1. *expr* is usually the global **status** variable.
2. **err_print** is most useful during debugging. The message it returns is probably not helpful to the user of your application.

Examples

```
    . . .
    IF status < 0 THEN
        CALL err_print(status)
    END IF
```

Related Functions

ERR_GET, ERR_QUIT, STARTLOG

ERR_QUIT

Overview

err_quit prints the INFORMIX-4GL error message corresponding to its argument on the Error Line and then terminates the program.

Syntax

CALL err_quit(*expr*)

Explanation

CALL	is a required keyword.
expr	is an integer expression.

Notes

1. *expr* is usually the global **status** variable.
2. **err_quit** is most useful during debugging. The message it returns is probably not helpful to the user of your application.

Examples

```
...
IF status < 0 THEN
    CALL err_quit(status)
END IF
```

Related Functions

ERR_GET, ERR_PRINT, STARTLOG

ERRORLOG

Overview

errorlog writes its argument to the current error log file.

Syntax

```
CALL errorlog(str)
```

Explanation

CALL	is a required keyword.
str	is a string constant or a CHAR variable.

Notes

1. The error log file is created by the **startlog** function.
2. If *str* is not a CHAR variable nor a string constant, INFORMIX-4GL will convert it to a string.
3. The primary use for the **errorlog** function is debugging and customized error handling.

Examples

```
CALL startlog("/usr/steve/error.log")
...
FUNCTION start_menu()
CALL errorlog("Entering start_menu function")
...
```

Related Functions

STARTLOG

INFIELD

Overview

The **infield** function tests whether its argument is the current screen field.

Syntax

```
infield(field-name)
```

Explanation

field-name is the name of a screen field.

Notes

1. **infield** is a Boolean function that returns the value true if *field-name* is the name of the current screen field and otherwise returns the value false.

2. Use **infield** during an INPUT or INPUT ARRAY statement to take field-dependent actions.

3. Outside of the INPUT or INPUT ARRAY statement, **infield** returns a value corresponding to the field that was current when the user terminated the INPUT or INPUT ARRAY statement.

Examples

The following example uses **infield** along with **showhelp** to give field-dependent help messages.

```
INPUT p_rec.* FROM sc_rec.*
    ON KEY(CONTROL-B)
        CASE
            WHEN infield(field1)
                CALL showhelp(101)
            WHEN infield(field2)
                CALL showhelp(102)
            WHEN infield(field3)
                CALL showhelp(103)
            ...
        END CASE
END INPUT
```

LENGTH

Overview

The **length** function returns the number of bytes in its string argument after deleting all trailing spaces.

Syntax

length(*str*)

Explanation

str	is a string constant or a CHAR variable.

Examples

The following example, taken from the FORMAT section of a report, centers a title on an 80-column page.

```
    LET title = "Invoice for ", fname CLIPPED,
                         " ", lname CLIPPED
    LET offset = (80 - length(title))/2
    PRINT COLUMN offset, title
```

NUM_ARGS

Overview

num_args returns the number of arguments on the command line.

Syntax

```
num_args( )
```

Notes

1. You may design your program to expect or allow arguments after the name of the executable program. The **num_args** function tells you how many arguments were on the line, in addition to the program name, and the **arg_val** function retrieves the individual arguments.

Examples

If your compiled, executable 4GL program name is **myprog** and you executed the command line

```
myprog john mary ellen joe frank
```

the following program segment puts the names you entered into an array of CHAR variables.

```
DEFINE args ARRAY[8] OF CHAR(10),
       i    SMALLINT

FOR i = 1 TO num_args()
    LET args[i] = arg_val(i)
END FOR
```

Related Functions

ARG__VAL

SCR_LINE

Overview

scr_line returns the number of the current screen row during an INPUT ARRAY statement.

Syntax

```
scr_line( )
```

Notes

1. The current screen row is the row where the cursor is located at the beginning of a BEFORE ROW or AFTER ROW clause.
2. The first row of both the program array and the screen array is numbered 1.
3. If you execute a statement in a BEFORE, AFTER, or ON KEY clause that changes the value of **arr_curr**, **scr_line**, or **arr_count**, the new value remains for the duration of the clause (unless changed again.) After INFORMIX-4GL executes the last statement in the clause, INFORMIX-4GL restores the original value.

Examples

The following example tests the user input and rejects it if the customer is not from California. (See the definition of **arr_curr()**.)

```
DEFINE p_array ARRAY[90] OF RECORD
    fname    CHAR(15),
    lname    CHAR(15),
    state    CHAR(2)
    END RECORD,
    pa_curr,
    sc_curr SMALLINT

INPUT ARRAY p_array FROM scr_array.*
    AFTER FIELD state
        LET pa_curr = arr_curr()
        LET sc_curr = scr_line()
        IF upshift(p_array[pa_curr].state) != "CA" THEN
            ERROR "Customers must be from California"
            INITIALIZE p_array[pa_curr].* TO NULL
            CLEAR scr_array[sc_curr].*
            NEXT FIELD fname
        END IF
END INPUT
```

Related Functions

ARR_COUNT, ARR_CURR

SET_COUNT

Overview

set_count tells INFORMIX-4GL the number of filled rows in a program array.

Syntax

```
CALL set_count(expr)
```

Explanation

CALL	is a required keyword.
expr	is an integer expression.

Notes

1. Before you use the INPUT ARRAY WITHOUT DEFAULTS or the DISPLAY ARRAY statement, you must call the **set_count** function with an argument that is the total number of filled rows in the program array. This allows the program to know the initial value to return for **arr_count**.

Examples

```
CALL set_count(23)
INPUT ARRAY p_array WITHOUT DEFAULTS
    FROM s_array.*
```

Related Functions

ARR_COUNT

SHOWHELP

Overview

showhelp displays a help screen and restores the previous screen when completed.

Syntax

```
CALL showhelp(nexpr)
```

Explanation

CALL	is a required keyword.
nexpr	is an integer expression.

Notes

1. When called with an argument that is the number of a help message in the help file named in an OPTIONS statement, **showhelp** clears the screen, displays the help message, and presents the user with a menu of help options. When the user selects **Resume** from the help menu, the previous screen is restored.

2. See Appendix G for assistance in setting up a help file.

Examples

The following example uses **infield** along with **showhelp** to give field-dependent help messages.

```
INPUT p_rec.* FROM sc_rec.*
    ON KEY(CONTROL-B)
        CASE
            WHEN infield(field1)
                CALL showhelp(101)
            WHEN infield(field2)
                CALL showhelp(102)
            WHEN infield(field3)
                CALL showhelp(103)
            ...
        END CASE
END INPUT
```

STARTLOG

Overview

startlog opens an error log file.

Syntax

```
CALL startlog(filename)
```

Explanation

CALL	is a required keyword.
filename	is a quoted string or CHAR variable that evaluates to the name of the error log file.

Notes

1. If the file specified by *filename* does not exist, **startlog** will create it. If the file does exist, **startlog** will open the file and position the file pointer so that subsequent errors will be appended to the end of the file.

2. If you do not want the error log file to reside in the current directory, you must specify a full pathname.

3. After you call the **startlog** function, a record of every subsequent error that occurs during the execution of your 4GL program will be written to the error log file.

4. The error record will consist of the date, the time, the source module name and line number, the error number, and the error message.

5. You may write your own messages to the error log file by using the **errorlog** function.

Examples

```
CALL startlog("/usr/steve/error.log")
...
FUNCTION start_menu()
CALL errorlog("Entering start_menu function")
...
```

Related Functions

ERRORLOG

UPSHIFT

Overview

upshift returns a string in which all lowercase characters in its argument are converted to uppercase.

Syntax

upshift(*str*)

Explanation

str	is a string constant or a variable of type CHAR.

Notes

1. Non-alphabetic characters in *str* are not altered.
2. You can use the **upshift** function in an expression (when such usage is allowed) or assign the value returned by the function to a varaible.

Examples

```
    LET u_str = upshift(str)

    DISPLAY u_str
```

Related Functions

DOWNSHIFT

Chapter 6

Programmer's Environment

Chapter 6 Table of Contents

Chapter Overview 5

The INFORMIX-4GL Menu 6

The MODULE Design Menu 7
  The **Modify** Option 7
  The **New** Option 10
  The **Compile** Option 10
  The **Program_Compile** Option 11
  The **Run** Option 11
  The **Exit** Option 12

The FORM Design Menu 12
  The **Modify** Option 12
  The **Generate** Option 14
  The **New** Option 14
  The **Compile** Option 15
  The **Exit** Option 15

The PROGRAM Design Menu 16
  The **Modify** Option 16
  The **New** Option 19
  The **Compile** Option 19
  The **Planned_Compile** Option 20
  The **Run** Option 21
  The **Drop** Option 21
  The **Exit** Option 22

The QUERY LANGUAGE Menu 22

Chapter Overview

INFORMIX-4GL provides a Programmer's Environment that consists of a series of nested menus. The philosophy behind the menu structure is that you should be able to move among the various steps of program development and the design of screen forms in a way that supports the design process. The INFORMIX-4GL Programmer's Environment anticipates the steps of development and keeps track of the components of your application. To enter the Programmer's Environment, type `i4gl` in response to the system prompt.

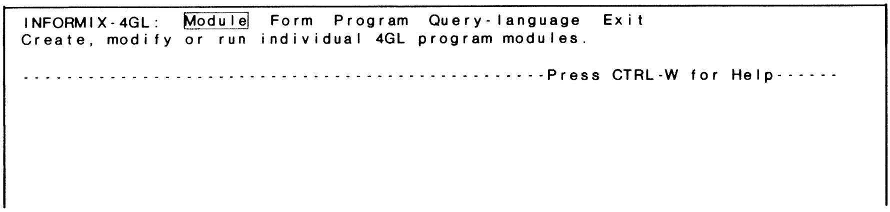
If you desire, you can invoke the compilers and editor directly from the operating system prompt and not use the Programmer's Environment. See Chapter 1 for a description of the proper command line syntax.

The INFORMIX-4GL Menu

At the top level, you have five options:

Module	Work on an INFORMIX-4GL program module.
Form	Work on a screen form.
Program	Describe components of a multi-module program.
Query_language	Use the RDSQL interactive interface.
Exit	Return to the operating system.

Putting these options at the top anticipates that you enter the INFORMIX-4GL environment with one of the three development areas in mind.



```
INFORMIX-4GL: Module  Form  Program  Query-language  Exit
Create, modify or run individual 4GL program modules.

---------------------------------------------Press CTRL-W for Help------
```

You select an option from the menu in either of two ways: by typing the first letter of the option or by using the SPACEBAR or ARROW keys to move the cursor to the option you want and pressing RETURN.

The MODULE Design Menu

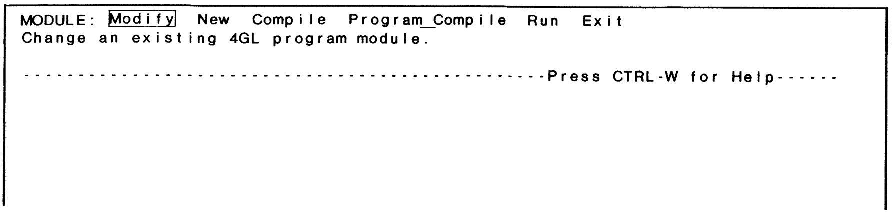
There are two kinds of program files: main modules (containing at least the MAIN routine) and auxiliary modules (containing FUNCTION or REPORT routines only). You select the **Module** option to work on either of these.

After you select the **Module** option, you have the following options:

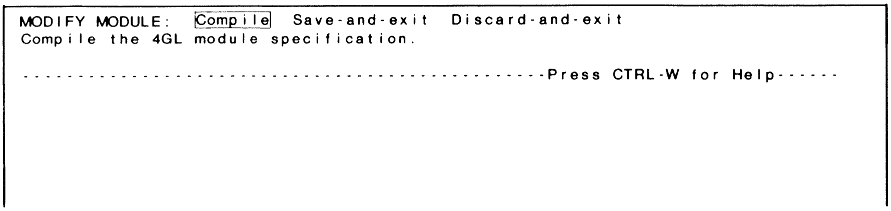
Modify	Change an existing 4GL program module.
New	Create a new 4GL program module.
Compile	Compile an existing 4GL program module.
Program_Compile	Compile a 4GL application program.
Run	Run an existing 4GL program module or application program.
Exit	Return to the INFORMIX-4GL Menu.

The Modify *Option*

Make changes to an existing 4GL program module.

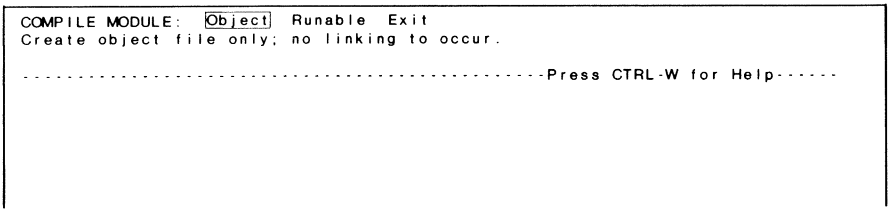

```
MODULE: Modify  New  Compile  Program_Compile  Run  Exit
Change an existing 4GL program module.

-----------------------------------------------Press CTRL-W for Help------
```

If you select this option, **INFORMIX-4GL** requests a name for the file and then prompts you to name your editor. If you have designated an editor with the DBEDIT environment variable (see Appendix C) or named an editor previously in this session, **INFORMIX-4GL** takes you immediately to your editor with the file that you named. The file being edited becomes the *current file.* When you leave the editor, **INFORMIX-4GL** moves to the MODIFY MODULE Menu with the **Compile** option highlighted.



```
MODIFY MODULE:  Compile  Save-and-exit  Discard-and-exit
Compile the 4GL module specification.

--------------------------------------------------Press CTRL-W for Help------
```

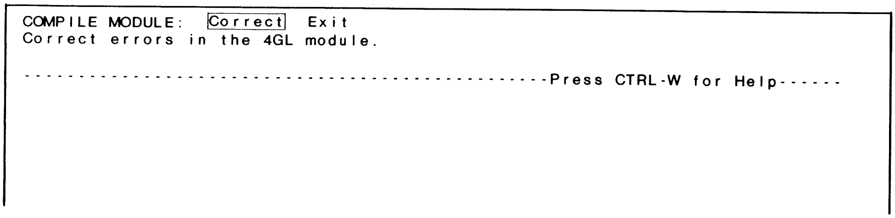
If you select the **Compile** option, **INFORMIX-4GL** displays the COMPILE MODULE Menu:



```
COMPILE MODULE:  Object  Runable  Exit
Create object file only; no linking to occur.

--------------------------------------------------Press CTRL-W for Help------
```

The **Object** option creates a **.o** file and makes no attempt to link the file with other files. The **Runable** option assumes that the current program module is a complete 4GL program and that no other module needs to be linked to it. If this is not the case, then you should use the **Object** option.

If there are compilation errors, INFORMIX-4GL presents the following menu:



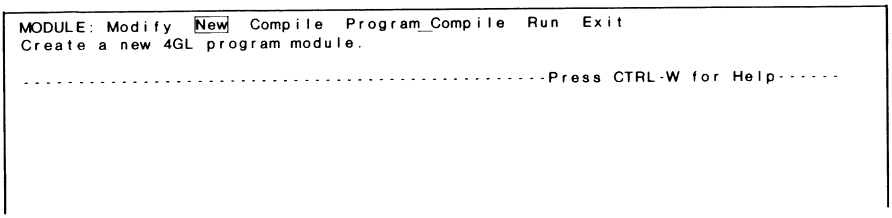
```
COMPILE MODULE:  Correct  Exit
Correct errors in the 4GL module.

--------------------------------------------Press CTRL-W for Help------
```

If you choose to correct the errors, INFORMIX-4GL puts you into your editor with a copy of your source module containing embedded error messages. Make corrections to your source file, save your changes, and leave your editor. You do not have to erase the error messages; INFORMIX-4GL will do it for you. You return to the previous MODIFY MODULE Menu to choose whether to compile or to save or discard your changes.

If there are no compilation errors, INFORMIX-4GL presents the MODIFY MODULE Menu with the **Save-and-Exit** option highlighted. If you select this option, INFORMIX-4GL writes the current file (the source file) to disk with the filename extension **.4gl**. It writes the object file to disk with the extension **.o**, or writes the executable file to disk with the filename extension **.4ge**. If you select the **Discard-and-Exit** option, INFORMIX-4GL discards any changes made since you selected the **Modify** option and leaves your files as they were previously.

The New *Option*

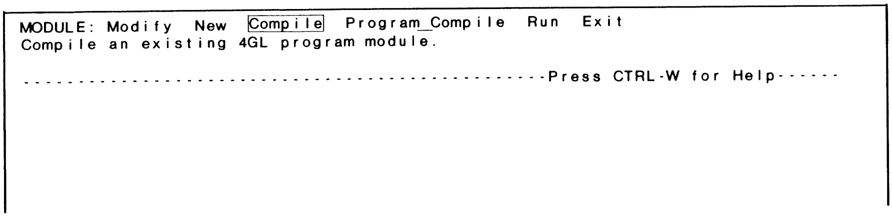
Create a new main or auxiliary module. This option is similar to the **Modify** option, except that the submenu is titled NEW MODULE. INFORMIX-4GL does not list the names of existing modules, but asks you to enter a new module name instead. If you have not designated an editor with DBEDIT or named an editor previously in this session, INFORMIX-4GL prompts you for one. It then puts you into the editor.



```
MODULE: Modify  New  Compile  Program_Compile  Run  Exit
Create a new 4GL program module.
---------------------------------------------Press CTRL-W for Help------
```

The Compile *Option*

Compile a program module. This option allows you to compile a program module without having first to select the **Modify** option.



```
MODULE: Modify  New  Compile  Program_Compile  Run  Exit
Compile an existing 4GL program module.
---------------------------------------------Press CTRL-W for Help------
```

After receiving the name of the program module to compile, INFORMIX-4GL goes to the COMPILE MODULE Menu that gives you the options of **Object**, **Runable**, or **Exit**. From there the situation is the same as described earlier for the **Modify** option.

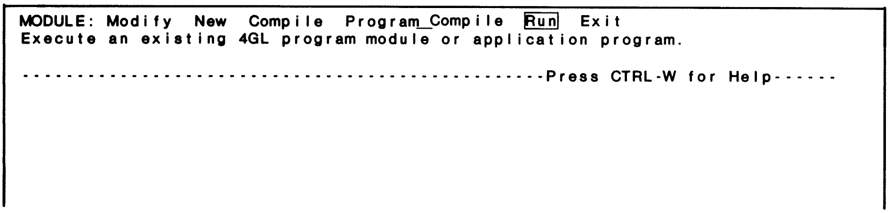
The **Program_Compile** *Option*

The **Program_Compile** option of the MODULE Menu is the same as the **Compile** option of the PROGRAM Menu (see that option for details). It permits you to compile and link modules as described in the program specification database, taking into account the time when the modules were last updated. This option is useful when you have just modified a single module of a complex program and want to test it by compiling and linking it with the other modules.

Note: If you are running a DOS system on MS-NET, there are certain procedures you must follow if you want to compile an INFORMIX-4GL program. See the section "Compiling and Executing Programs" in Chapter 1 of the INFORMIX-4GL Reference Manual for information on how to do this.

The **Run** *Option*

Run a compiled program.

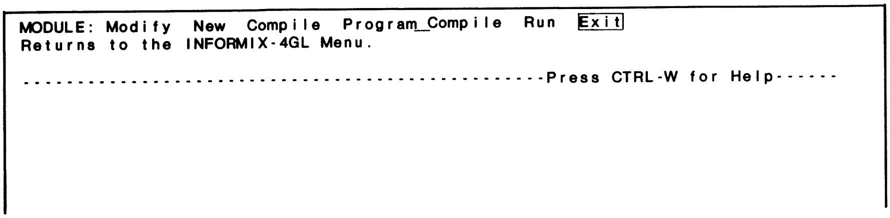


```
MODULE: Modify  New  Compile  Program_Compile  [✓ Run]  Exit
Execute an existing 4GL program module or application program.

--------------------------------------------Press CTRL-W for Help------
```

The RUN PROGRAM screen presents a list of compiled modules (which could be a program) with the highlight on the module corresponding to the current file, if any. A compiled program must have the extension **.4ge** to be included in the list. If you compile a program outside the Programmer's Environment and want it to appear in this list in the future, give the executable file the extension **.4ge**. If no compiled programs exist, INFORMIX-4GL prints an error message and returns you to the MODULE Menu.

The Exit *Option*

Return to the INFORMIX-4GL Menu.



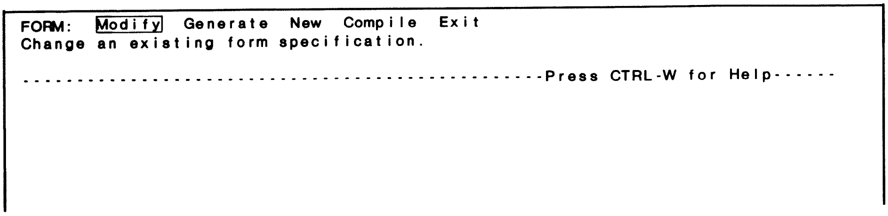
```
MODULE: Modify  New  Compile  Program_Compile  Run  [Exit]
Returns to the INFORMIX-4GL Menu.
--------------------------------------------Press CTRL-W for Help------
```

The FORM Design Menu

You can generate, modify, and compile screen forms through the FORM Menu. Select the **Form** option from the INFORMIX-4GL Menu to view the FORM Menu.

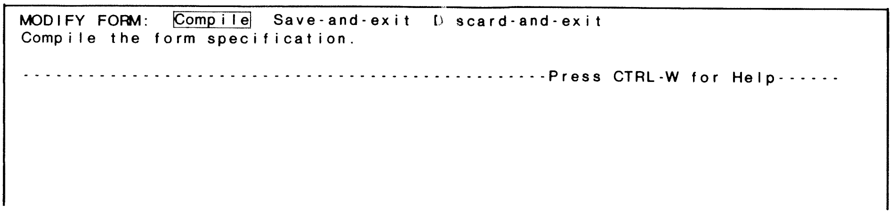
The Modify *Option*

The **Modify** option of the FORM Menu is very similar to that option in the MODULE Menu.



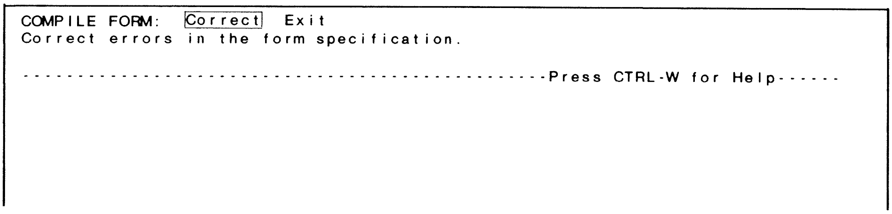
```
FORM:  [Modify]  Generate  New  Compile  Exit
Change an existing form specification.
--------------------------------------------Press CTRL-W for Help------
```

If you select this option, the Programmer's Environment asks you for the name of the form specification file you want to modify and places you in the editor. When you leave the editor, INFORMIX-4GL presents you with the MODIFY FORM Menu with the **Compile** option highlighted.



```
MODIFY FORM:  Compile  Save-and-exit  Discard-and-exit
Compile the form specification.

----------------------------------------------Press CTRL-W for Help------
```

If there are compilation errors, INFORMIX-4GL presents you with the COMPILE FORM Menu:



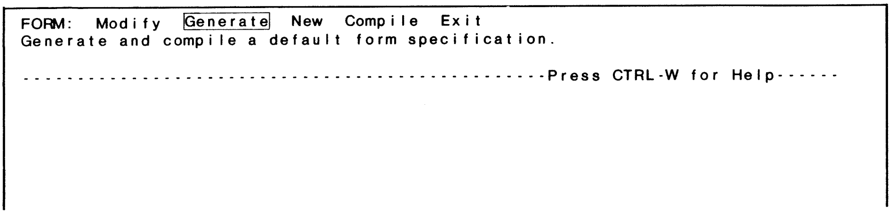
```
COMPILE FORM:  Correct  Exit
Correct errors in the form specification.

----------------------------------------------Press CTRL-W for Help------
```

After you correct your errors (you do not have to delete the error messages), INFORMIX-4GL returns you to the MODIFY FORM Menu with the **Compile** option highlighted.

If there were no compilation errors, you may choose whether to save the modified form specification file and the compiled form or to discard the changes and return to the state before you chose to modify the form specification file.

The Generate *Option*

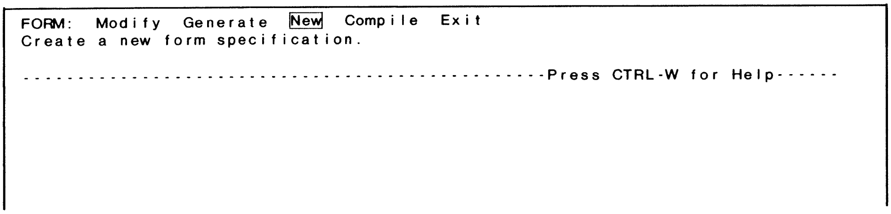
When you select the **Generate** option, **INFORMIX-4GL** asks you to select a database, choose a name for the form, and choose the names of tables. **INFORMIX-4GL** then runs the **-d** option of **FORM4GL** to create and compile a default form.



```
FORM:  Modify  [✓ Generate]  New  Compile  Exit
Generate and compile a default form specification.

-----------------------------------------------Press CTRL-W for Help------
```

The New *Option*

The **New** option of the FORM Menu allows you to create a form from scratch without going through the **Generate** option.

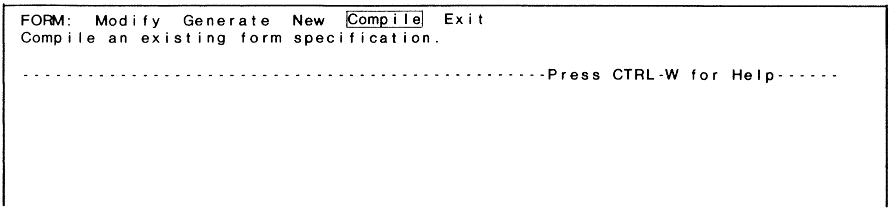


```
FORM:  Modify  Generate  [✓ New]  Compile  Exit
Create a new form specification.

-----------------------------------------------Press CTRL-W for Help------
```

After asking you for the name of your form specification file, **INFORMIX-4GL** places you in the editor where you can create a form specification file. When you leave the editor, **INFORMIX-4GL** transfers you to the NEW FORM Menu that is like the MODIFY FORM Menu. You may compile your form and correct it in the same way.

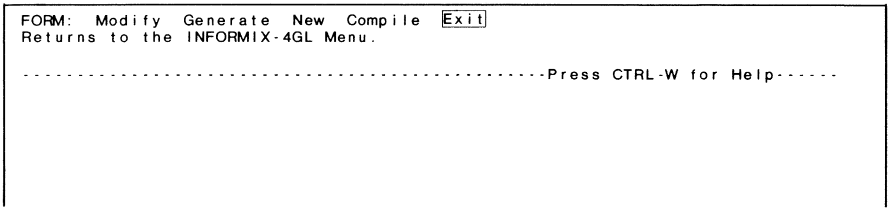
The Compile *Option*

The **Compile** option allows you to compile an existing form specification file without having to go through the **Modify** option. INFORMIX-4GL asks you for the name of the form specification file and then performs the compilation. If compilation is not successful, INFORMIX-4GL presents the COMPILE FORM Menu with the highlight on the **Correct** option.


```
FORM:  Modify  Generate  New  [Compile]  Exit
Compile an existing form specification.

---------------------------------------------Press CTRL-W for Help------
```

The Exit *Option*

The **Exit** option returns you to the INFORMIX-4GL Menu.


```
FORM:  Modify  Generate  New  Compile  [Exit]
Returns to the INFORMIX-4GL Menu.

---------------------------------------------Press CTRL-W for Help------
```

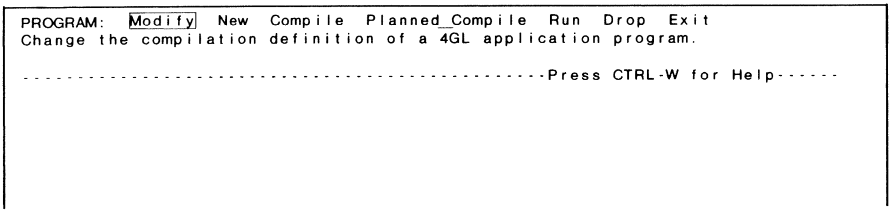
The PROGRAM Design Menu

If you select the **Program** option from the INFORMIX-4GL Menu, INFORMIX-4GL searches your DBPATH (see Appendix C) to determine whether the program specification database (**syspgm4gl**) exists. The program specification database describes the modules and compiler options that constitute your entire INFORMIX-4GL program. If the database does not exist, INFORMIX-4GL asks you whether you want to create one. If you enter y in response to the prompt, INFORMIX-4GL creates the **syspgm4gl** database, grants CONNECT privilege to PUBLIC, and presents you with the PROGRAM Menu. As Database Administrator of **syspgm4gl**, you may restrict other users' access later. If **syspgm4gl** already exists, INFORMIX-4GL takes you immediately to the PROGRAM Menu.

From the PROGRAM Menu you can create or modify information in a program specification, compile and link your program, and run your program.

The **Modify** *Option*

The **Modify** option allows you to modify the data in an existing program specification.



```
PROGRAM:  Modify  New  Compile  Planned_Compile  Run  Drop  Exit
Change the compilation definition of a 4GL application program.

---------------------------------------------Press CTRL-W for Help------
```

Note: You can select the **Modify** option only if a program specification exists. If no program specification exists, you may create one by selecting the **New** option.

INFORMIX-4GL asks you to indicate the name of the program specification you want to modify. It then invokes a screen and menu allowing you to update the information in the program specification database:

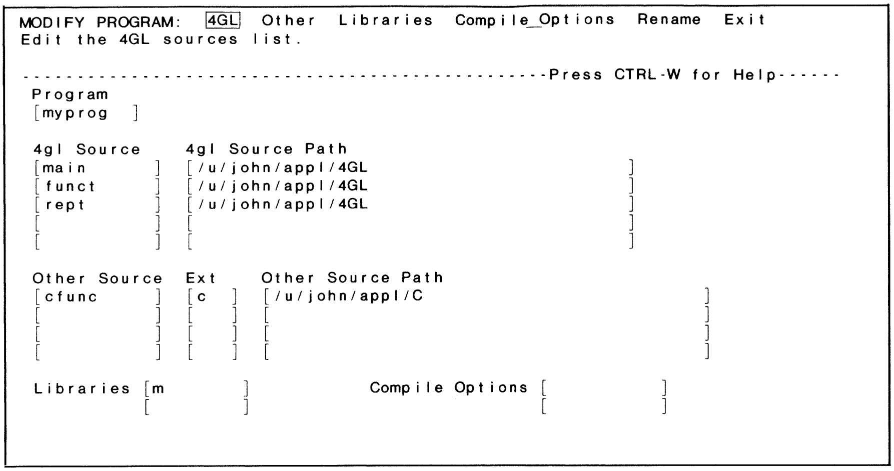

```
MODIFY PROGRAM:  4GL  Other  Libraries  Compile_Options  Rename  Exit
Edit the 4GL sources list.

-------------------------------------------Press CTRL-W for Help------
 Program
 [myprog  ]

 4gl Source     4gl Source Path
 [main       ]  [/u/john/appl/4GL                        ]
 [funct      ]  [/u/john/appl/4GL                        ]
 [rept       ]  [/u/john/appl/4GL                        ]
 [           ]  [                                        ]
 [           ]  [                                        ]

 Other Source   Ext    Other Source Path
 [cfunc      ]  [c  ]  [/u/john/appl/C                        ]
 [           ]  [   ]  [                                      ]
 [           ]  [   ]  [                                      ]
 [           ]  [   ]  [                                      ]

 Libraries [m          ]          Compile Options [          ]
           [           ]                          [          ]
```

Figure 6-1. Sample Entry in Program Specification

In Figure 6-1, you named the program **myprog**. You could change this name by selecting the **Rename** option. The significance of the program name is that the executable program resulting from compiling and linking all the source files and libraries has the program name with the extension **.4ge**. In this case, the executable program has the name **myprog.4ge**.

The **4GL** option allows you to update the entries for 4GL Source and 4GL Source Path. The five rows of fields under these labels form a screen array and, when you select the **4GL** option, INFORMIX-4GL executes an INPUT ARRAY statement to allow you to move through the array (and to scroll for entries up to a maximum of 30). See the INPUT ARRAY statement in Chapter 7 for how to use your function keys to scroll, delete rows, and to insert new rows (you cannot redefine the function keys as you can with a 4GL program).

In the example shown in Figure 6-1, you have chosen to break up your 4GL source program into three modules: a module containing your main program (**main.4gl**), a module containing functions (**funct.4gl**), and a module containing REPORT routines (**rept.4gl**). These modules are all located in the directory **/u/john/appl/4GL**. If you also have a module containing only global variables (**global.4gl**), you must list that module here.

The **Other** option allows you to update the three-column screen array with headings Other Source, Ext, and Other Source Path. This is where you place the filename and location of non-4GL source or object code, either using INFORMIX-ESQL/C or C. In the example shown in Figure 6-1, you listed a file containing C function source code (**cfunc.c**) located in **/u/john/appl/C**. You may list up to 30 files in this array.

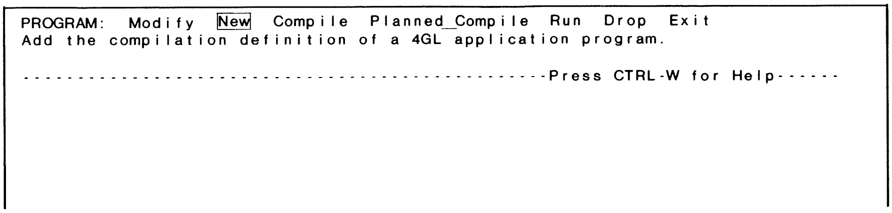
The **Libraries** option permits you to add the name of up to ten special libraries that you want to link with your program. INFORMIX-4GL calls the C compiler to do the linking and adds the appropriate **–l** prefix, so you should enter only what follows the prefix. In Figure 6-1, you have requested only the standard C math library.

The **Compile_Options** option lets you supply up to ten C compile options if desired. You may not enter the **–e** or **–a** options of **c4gl** (see Chapter 1).

The **Exit** option returns you to the PROGRAM Menu.

The New *Option*

The **New** option of the PROGRAM Menu allows you to create a new specification of the program modules and libraries that make up the desired application program. You may also specify any necessary compiler or loader options.

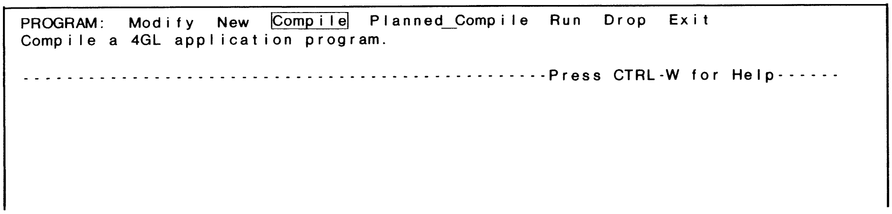


```
PROGRAM:  Modify  New  Compile  Planned_Compile  Run  Drop  Exit
Add the compilation definition of a 4GL application program.
---------------------------------------------Press CTRL-W for Help------
```

The **New** option is identical to the **Modify** option, except that you must first supply a name for your program. It then puts you with a blank form into the NEW PROGRAM Menu that has the same options as the MODIFY PROGRAM Menu.

The Compile *Option*

The **Compile** option performs the compilation and linking described by the program specification database, taking into account the time when files were last updated. It compiles only those files that been changed since they were last compiled.

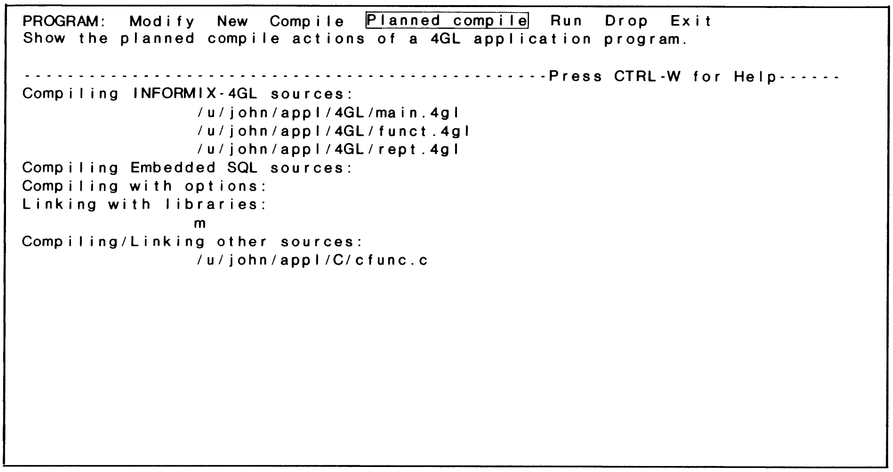


```
PROGRAM:  Modify  New  Compile  Planned_Compile  Run  Drop  Exit
Compile a 4GL application program.
---------------------------------------------Press CTRL-W for Help------
```

INFORMIX-4GL lists each step of the compilation as it occurs. See the screen in the following section.

The **Planned_Compile** *Option*

Taking into account the time of last change for the various files in the dependency relationships, the **Planned_Compile** option prompts for a program name and presents you with a summary of the steps that will be executed if you select **Compile**. No compilation actually takes place.

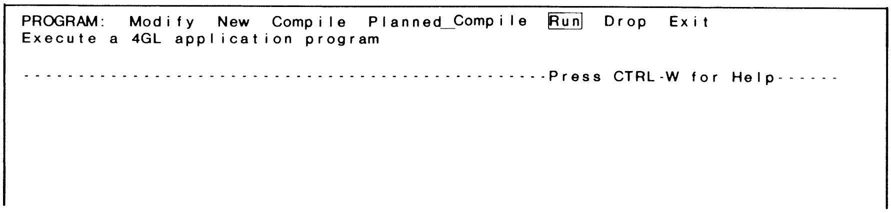


```
PROGRAM:  Modify  New  Compile  Planned compile  Run  Drop  Exit
Show the planned compile actions of a 4GL application program.

----------------------------------------------Press CTRL-W for Help------
Compiling INFORMIX-4GL sources:
                /u/john/appl/4GL/main.4gl
                /u/john/appl/4GL/funct.4gl
                /u/john/appl/4GL/rept.4gl
Compiling Embedded SQL sources:
Compiling with options:
Linking with libraries:
                m
Compiling/Linking other sources:
                /u/john/appl/C/cfunc.c
```

If you have made changes in all the components of the program listed in Figure 6-1 since the last time you compiled them, INFORMIX-4GL presents the previous screen.

The **Run** *Option*

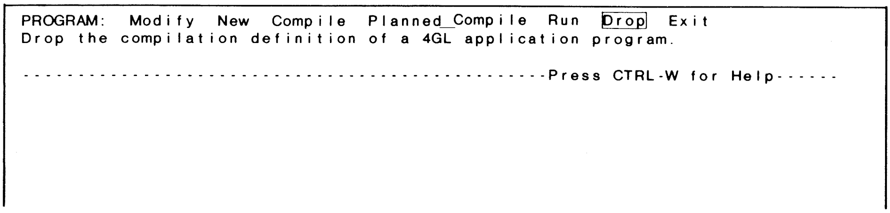
The **Run** option of the PROGRAM Menu is the same as the **Run** option of the MODULE Menu. It presents you with a list of compiled programs (with the extension **.4ge**) and places the highlight on the current program, if any. It then executes the program you select.



```
PROGRAM:  Modify  New  Compile  Planned_Compile  Run  Drop  Exit
Execute a 4GL application program

----------------------------------------------Press CTRL-W for Help------
```

The **Drop** *Option*

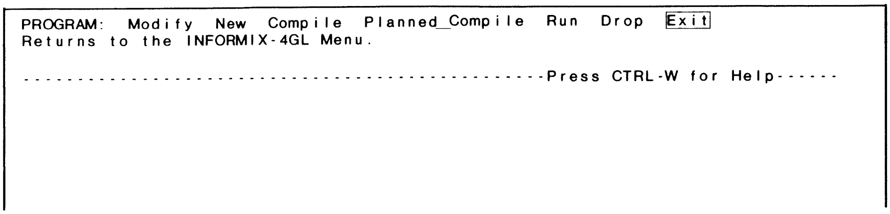
The **Drop** option of the PROGRAM Menu prompts you for a program name and removes the compilation definition for that program from the **syspgm4gl** database. Your program and 4GL modules are *not* removed.



```
PROGRAM:  Modify  New  Compile  Planned_Compile  Run  Drop  Exit
Drop the compilation definition of a 4GL application program.

----------------------------------------------Press CTRL-W for Help------
```

The Exit *Option*

The **Exit** option of the PROGRAM Menu returns you to the INFORMIX-4GL Menu.



```
PROGRAM:  Modify  New  Compile  Planned_Compile  Run  Drop  [Exit]
Returns to the INFORMIX-4GL Menu.

-------------------------------------------------Press CTRL-W for Help------
```

The QUERY LANGUAGE Menu

The RDSQL interactive interface is identical to the interactive SQL interface of INFORMIX-SQL. You can exercise this option only if you have separately purchased and installed INFORMIX-SQL on your system. The **Query-language** option is placed at the top level to allow you to test RDSQL statements without leaving the INFORMIX-4GL environment. You can also create, execute and save SQL scripts.