INFORMIX®

INFORMIX-4GL

A Twenty-Minute Guide

Version 1.10

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

November 1, 1986
Version 1.10

Design and Production
Clennon Associates

Table of Contents

INFORMIX-4GL

A Twenty-Minute Guide

Version 1.10

2

Building an information management application with a database product should not require an advanced degree in computer science. You should be able to move quickly from the conception of a series of operations you would like to perform on the database to an application program that allows you and others to perform these operations. At the same time, you want the interface between your application and the user to be friendly and easy to use. Creating windows, menus, and screen forms that facilitate the user's task of entering and retrieving data should be easy to do. Extracting information and displaying it in an attractive report format should be straightforward and part of the same program. You should be able to execute all of these operations—and more, using an industry standard retrieval syntax. Other people interested in your application should be able to read and understand your program without extensive training.

Informix Software, Inc., has simplified the process of creating an application by combining natural database operations with program-flow statements into a fourth-generation language, INFORMIX-4GL. INFORMIX-4GL is built upon RDSQL™, our extension of the Structured Query Language (SQL) developed by IBM. SQL is rapidly becoming the standard query language for database management systems, and RDSQL conforms to the ANSI standards for SQL implementations.

INFORMIX-4GL joins five other members of the Informix product line and is completely compatible with each of these products:

- INFORMIX-SQL
- INFORMIX-ESQL/C (Embedded SQL for C)
- INFORMIX-ESQL/COBOL (Embedded SQL for COBOL)
- *File-it!*®
- C-ISAM™

Introduction

3

A well designed fourth-generation language has two characteristics that make it desirable as an application-making tool. The first is the presence of non-procedural statements that permit you to describe what is wanted without having to list the detailed steps on how it should be achieved. The second is that the language was developed with specific types of applications in mind—database applications. Unlike third-generation languages like C, Pascal, and COBOL that are generalized and have no particular application built into their design, a fourth-generation language has a specific focus. INFORMIX-4GL has been designed by experts in database management software. They have introduced features that make it simple for you to create powerful database applications with a few brief statements. With these statements you can perform the following functions:

- Use windows
- Create menus
- Collect input from screen forms
- Use SQL to manipulate a database
- Call for help screens
- Create reports
- Collect multi-row data from a single form with scrolling
- Provide query-by-example forms
- Trap user-entered function and control keys
- Set up conditional screen attributes
- Have access to debugging tools
- Call 4GL or C library functions

Menus

INFORMIX-4GL makes it easy to create Lotus-like ring menus (Figure 1) that list the possible options. The user can select an option by typing the first letter of the option or by using the [SPACEBAR] to move the highlight from one option to the next and pressing [RETURN].

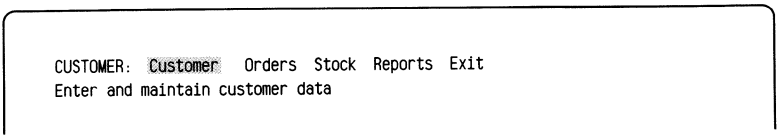

Figure 1. Ring Menu

Help Screens

You can enter help messages for each menu option and data entry opportunity into an ordinary text file. These messages are automatically read and displayed by your program at runtime when the user presses the help key. You can amend or enhance these help messages without recompiling your program.

Input from Screen Forms

You can create screen forms that check input data for a variety of integrity constraints (Figure 2). Your program collects data from the screen and puts it into program variables with a single statement. The user can move around the screen from field to field with the arrow keys or by using the [RETURN] key. The user signals when data entry is complete by pressing the ACCEPT key.

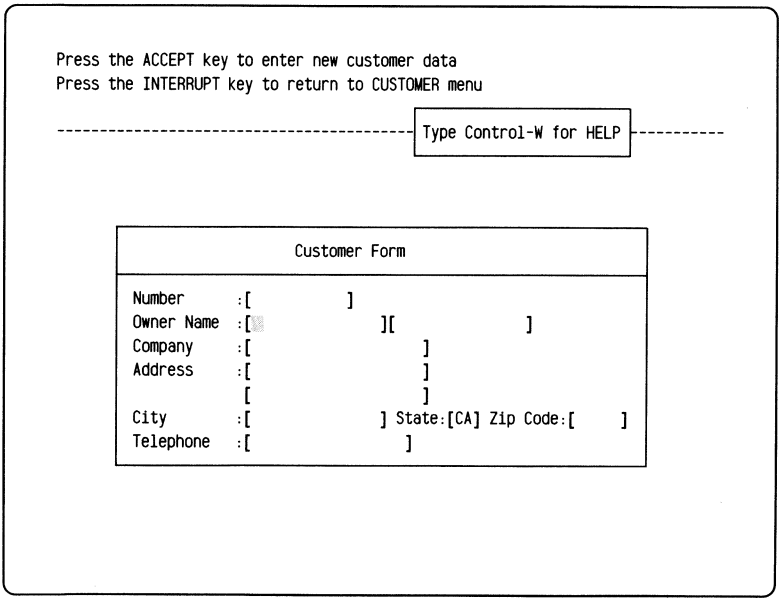

Figure 2. Screen at the Beginning of Data Entry

Input to Screen Arrays

You can collect data from multiple rows on a screen, called a "screen array." This allows the user to enter and update several rows at a time (Figure 3). For example, the user could enter all the items for an order, having the data scroll automatically when the screen array is full. Just a single line of 4GL code permits the user to move throughout the screen array, entering and changing data, inserting new rows or deleting old rows, and paging through the data that automatically gets stored in a program array.

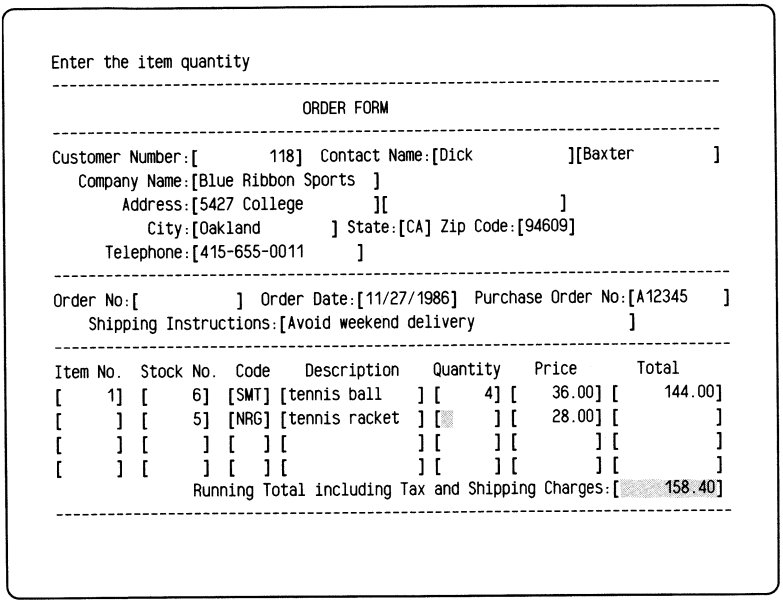

Figure 3. Input to a Screen Array

6

Window Management

With INFORMIX-4GL, you can create applications that devote different rectangular parts of the screen to different activities. Each rectangular portion of the screen is called a window. In your application programs, you can use windows to display screen forms, prompts, menus, report output, or anything else that you want the user to pay special attention to.

INFORMIX-4GL includes powerful window management statements that allow you to open, change, clear, and close windows within your programs. For example, the customer-entry program in this guide opens a state selection window if the user does not enter a valid state code in the State field of the Customer Form.

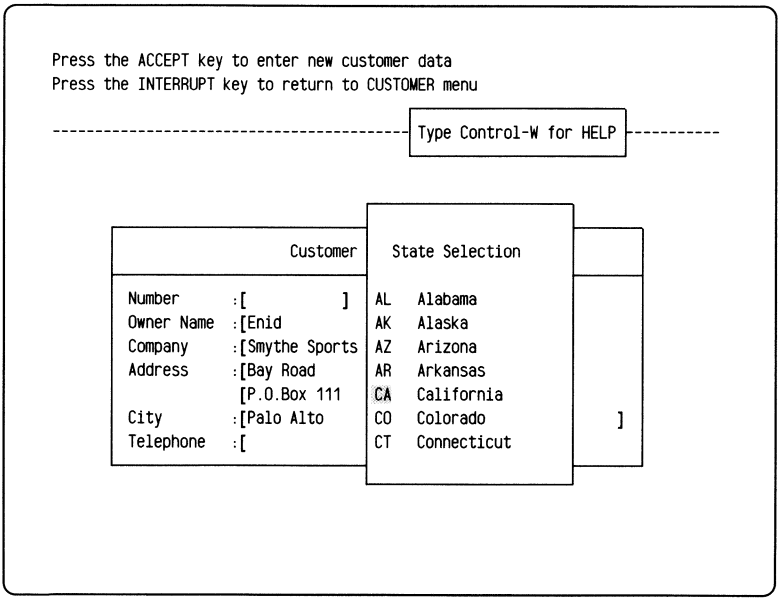

Figure 4. State Selection Window

When the user selects a state by positioning the cursor on a row and pressing the ACCEPT key, the program automatically displays the state code in the State field on the underlying form.

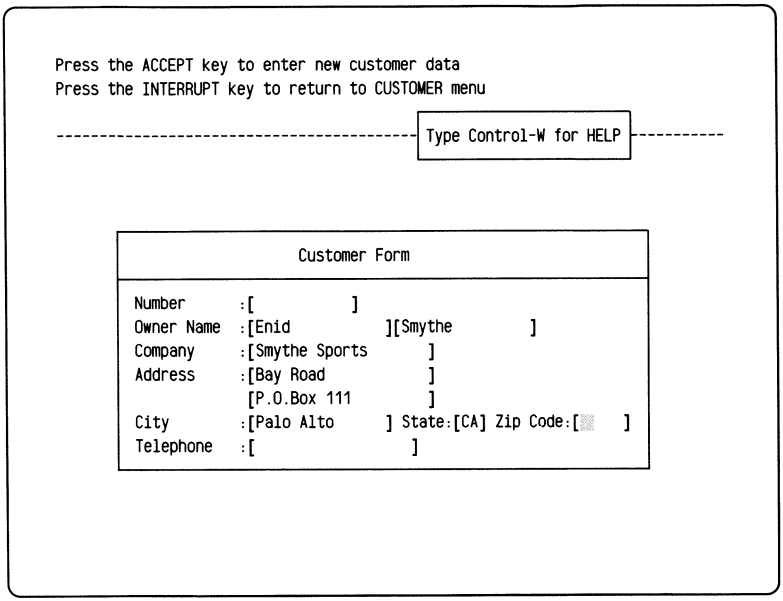

Figure 5. Automatic Entry of State Code in the State Field

With the window management statements provided by INFORMIX-4GL, you can create an effective user interface for your application programs.

Active Function Keys

Your program can capture user-entered function keys and control characters during data entry to provide a variety of programmable special effects, including special-purpose windows, data validation, field-by-field help messages, default values, and a display of alternative inputs. For example, in an application dealing with personnel records, you could display different help screens for the **department** field, depending upon the value already entered for the **division** field.

Query by Example

A simple INFORMIX-4GL statement collects data from a query-by-example entry and inserts it into a string. You can then use the string to prepare a dynamic SQL statement that queries the database. The entry values from the following screen (Figure 6) cause INFORMIX-4GL to place the string

```
customer.customer_num)"115" and customer.lname matches "B*" and customer.city="Oakland"
```

in a specified program variable.

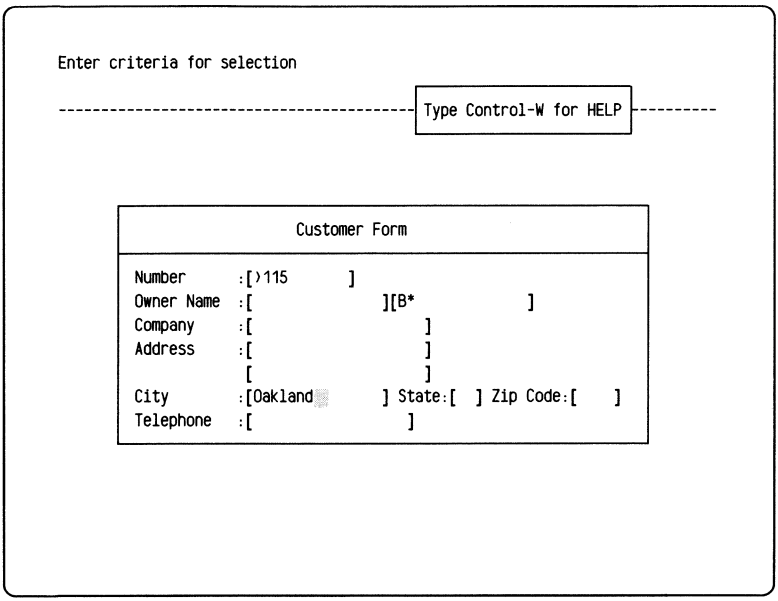

Figure 6. Query-by-Example Sample Entry

Reports
You can create reports that combine data from one or more tables as well as from computed program variables. You can execute other INFORMIX-4GL statements in the middle of the report. For example, your program can update your database in the middle of the report, if the intermediate results, calculated while printing the report, indicate that an update is appropriate.

Default Screen Attributes
You can set up data dictionary-based default screen display attributes and input-checking values and formats that will apply to all forms in your application. The attributes can be conditional, displaying a variable in green, for example, when its value is above 1000, in yellow when its value is less than 1000 but greater than 100, and in red when its value drops below 100.

SQL
With INFORMIX-4GL, you can use the full power of the ANSI standard SQL with Informix extensions to manipulate your database. This includes

- The SQL data manipulation language
- The SQL data definition language
- The full range of database-, table-, and column-level data security
- Transaction logging and recovery with commit and rollback
- The use of views as a convenience or to guarantee data integrity
- Clustered indexes and auto-indexing
- "Scrolling" select cursors and insert cursors that permit buffered insertion of data into a database
- The distinction between null values and zero values
- Outer joins of unlimited complexity
- Informix extensions to SQL that permit you to create, drop, and change databases in the middle of your application and to use powerful date functions in addition to the standard aggregates in SQL statements

10

Debugging Tools
The simplicity of a fourth-generation language reduces the length of your application and the time required to debug it. INFORMIX-4GL provides a number of debugging tools, including error logging and trapping and the recovery from errors.

Function Libraries
You can call functions from a supplied library of pre-compiled functions. You can also create your own functions using the C language or INFORMIX-4GL and call them from within your application.

And More
In addition to these features, INFORMIX-4GL contains a sophisticated application development interface, the Programmer's Environment. You or a team of developers can build an application through an interface that anticipates the steps of development and keeps track of the components of your application. After editing one module of your application, you can compile it and link it with the other modules of your application, and with whatever library functions you used—all with a single keystroke. Only those modules that have been altered since the last compilation are recompiled.

The full package of INFORMIX-4GL includes several utility programs. These programs ease the transition from previous applications that use Informix products, check and restore the integrity of your index files, load data from other sources, and manipulate the data dictionary tables that govern default attributes and data checking for your screen forms.

11

The quickest way to learn about the ease of programming with INFORMIX-4GL is to read through program excerpts that produce some of the effects that were described earlier. (These program excerpts are from the demonstration application that accompanies the product and that appears in Appendix A of the *INFORMIX-4GL Reference Manual.*) Since you signed on for only twenty minutes when you picked up this booklet, it is not possible to teach you all of the syntax for writing an INFORMIX-4GL program. You will be surprised, nevertheless, by how simple it is to read and understand the INFORMIX-4GL program excerpts that achieve these sophisticated effects.

Assume that you have been hired by a wholesaler of sports equipment to create a database management application to keep track of customers (retailers), the orders placed by these customers, and the types of stock, and to produce a series of reports based on this data. The examples that follow treat portions of this application. Assume that the database and its tables (files) have already been created. For example, one of the tables is the **customer** table and it has ten columns (fields) as shown in Figure 7.

Column Name	Data Type	Meaning of Data Type
customer_num	serial(101)	unique integers starting with 101
fname	char(15)	character string of length 15
lname	char(15)	character string of length 15
company	char(20)	character string of length 20
address1	char(20)	character string of length 20
address2	char(20)	character string of length 20
city	char(15)	character string of length 15
state	char(2)	character string of length 2
zipcode	char(5)	character string of length 5
phone	char(18)	character string of length 18

Figure 7. The **customer** Table

Example Application

Menus

You can produce the menu illustrated earlier in Figure 1 with the following code:

```
MAIN
    DEFER INTERRUPT
    OPTIONS
        HELP FILE "helpdemo"

    MENU "MAIN"
    COMMAND "Customer" "Enter and maintain customer data" HELP 101
        CALL cust()
    COMMAND "Orders" "Enter and maintain orders" HELP 102
        CALL ord()
    COMMAND "Stock" "Enter and maintain stock list" HELP 103
        CALL stock()
    COMMAND "Reports" "Print reports and mailing labels" HELP 104
        CALL rept()
    COMMAND "Exit" "Exit program and return to operating system" HELP 105
        CLEAR SCREEN
        EXIT PROGRAM
    END MENU
END MAIN
```

Figure 8. The MAIN Program Routine

This program excerpt embodies the main routine for the entire application. Every INFORMIX-4GL program must have a routine that starts with the keyword MAIN and ends with END MAIN. This is where program control starts when you run the program. The first line of the MAIN routine keeps the program from stopping immediately when the user presses the INTERRUPT key. The next two lines set the pathname of the file containing the HELP messages.

The menu is created by the non-procedural set of statements between MENU and END MENU. The name of the menu is written after the keyword MENU. Each option is listed after the keyword COMMAND and is followed by a string that is listed on the second line of the screen below the menu options when that option is highlighted (see Figure 1). Following the keyword HELP is the number of the help text that appears on the screen if the user presses the HELP key. Below each COMMAND line are a series of steps that the program follows if the user selects that particular option. In each case, except for the **Exit** option, the program calls a function and, when that function returns, redraws the menu automatically. You can write the functions to carry out any INFORMIX-4GL statements, including displaying a submenu.

The **Exit** option, in this case, clears the screen and returns the user to the operating system, using the EXIT PROGRAM statement.

Screen Forms

To create the screen form displayed in Figure 2, you must write a form specification file. Without leaving the Programmer's Environment you can create a default screen form with automatically generated field labels and modify it to produce **customer.per** as shown in Figure 9.

```
DATABASE stores

SCREEN
{

--------------------------------------- Type Control-W for HELP ----------


                              Customer Form

        Number       :[f000       ]
        Owner Name   :[f001           ][f002           ]
        Company      :[f003                ]
        Address      :[f004                ]
                      [f005                ]
        City         :[f006       ]    State:[a0]  Zip Code:[f007 ]
        Telephone    :[f008              ]

}

TABLES
customer

ATTRIBUTES
f000 = customer.customer_num, NOENTRY;
f001 = customer.fname;
f002 = customer.lname;
f003 = customer.company;
f004 = customer.address1;
f005 = customer.address2;
f006 = customer.city;
a0 = customer.state, UPSHIFT;
f007 = customer.zipcode;
f008 = customer.phone, PICTURE = "###-###-#### #####";
```

Figure 9. The **customer.per** Form Specification File

You label each field in the SCREEN section with a field tag and identify each field tag with the name of the field in the ATTRIBUTES section. In this case the field names are the same as the names of the columns in the **customer** table. You can also have fields that are not related to database columns. The field tags in this example were generated automatically using a "default screen" feature of the Programmer's Environment.

There are many display attributes that you can assign to each field. The **customer.per** example file uses only three. The field named **customer.customer_num** has the attribute NOENTRY, signifying that during input on the screen form, the cursor will not enter that field; it is for displaying values only. RDSQL assigns a unique value to **customer.customer_num** when you insert the row corresponding to the entries on the screen into the database. You do not want the user entering values in that field.

customer.per uses the UPSHIFT attribute to convert all entries in the **customer.state** field to uppercase characters. This way you can ensure that all state values are uppercase, even if the user did not enter them using uppercase letters.

Note: If the user does not enter a valid state code, the customer-entry program opens a state selection window that displays the states and corresponding abbreviations in a screen array. When the user moves the cursor to a state and presses the ACCEPT key, the program enters the state code in the State field automatically.

The form specification assigns the PICTURE attribute to the Telephone field. This attribute requires that an entry conform to a specified format. When the user moves the cursor into the Telephone field, INFORMIX-4GL displays the format and allows the user to enter only digits in the field.

```
Address     :[Bay Road             ]
             [P.O. Box 111         ]
City        :[Palo Alto      ] State:[CA] Zip Code:[94305]
Telephone   :[   -   -           ]
```

Figure 10. The PICTURE Attribute

After designing the form, you must compile it using the Form4GL program. If you work within the Programmer's Environment, this step occurs with one keystroke.

Entering Data Using a Screen Form

After creating a screen form, you will want to write the INFORMIX-4GL code to extract data from the screen and insert it into your database. Figure 11 shows program excerpts that do this.

```
GLOBALS
    DEFINE p_customer RECORD LIKE customer.*
    ...

END GLOBALS
...
    OPEN FORM customer FROM "customer"
    DISPLAY FORM customer
...
    FUNCTION add_cust()
    INPUT BY NAME p_customer.*
    LET p_customer.customer_num = 0
    INSERT INTO customer VALUES (p_customer.*)
END FUNCTION
```

Figure 11. Simple INPUT Example

The code at the top of Figure 11 defines a RECORD named **p_customer.** A record is a collection of variables of varying type that can be treated as a group. It corresponds closely to the Pascal or COBOL record and the C structure. In this case, **p_customer** is defined LIKE the **customer** table displayed in Figure 7. In this

brief statement, the code has defined ten variables with the same names and data types as the columns in the **customer** table. The code defines **p_customer** as a GLOBAL record, which means that all the functions in the INFORMIX-4GL program can use and alter the data stored in the variables of **p_customer.**

The next two lines of code are from the function that calls **add_cust.** They display the form described in Figure 9 and shown in Figure 2.

The function **add_cust** takes input from the **customer** screen and stores the data in the **customer** table of the database. The INPUT statement makes use of the fact that the field names on the form and the variables defined in **p_customer** are the same. The simplicity of the INPUT statement is matched only by its power. The user can move about the form using arrow keys or the [RETURN] key, filling in or rewriting any or all of the fields. When the entries are satisfactory, the user can complete the entry by pressing the ACCEPT key anywhere in the form or by pressing [RETURN] in the Telephone field.

The next line of **add_cust** handles the insertion of data entered on the form into the database. Setting **p_customer.customer_num** to zero signals RDSQL to provide the next sequential value to the **customer_num** column during the INSERT statement.

Although this program excerpt has the essential functionality you want, the full application embellishes it with additional features. The code in Figure 12 replaces the function **add_cust** with a more robust function **input_cust.**

```
FUNCTION input_cust()
    DISPLAY "Press the ACCEPT key to enter a new customer" AT 1,1
    DISPLAY "Press INTERRUPT to return to CUSTOMER menu" AT 2,1
    LET int_flag = FALSE          # Initialize 4GL's interrupt flag

# Section 1 - Collect data from form

   INPUT BY NAME p_customer.*
       AFTER FIELD state
           CALL statehelp()
   END INPUT
   IF int_flag THEN               # Test for interrupt
       LET int_flag = FALSE
       ERROR "Customer data discarded"
       RETURN FALSE
   END IF

# Section 2 - Insert data into database

    LET p_customer.customer_num = 0
    INSERT INTO customer VALUES (p_customer.*)
    LET p_customer.customer_num = SQLCA.SQLERRD[2]
    DISPLAY BY NAME p_customer.customer_num
    DISPLAY "Customer data entered" AT 24,1
    RETURN TRUE
END FUNCTION
```

Figure 12. Full Routine Using the INPUT Statement

input_cust consists of two main sections. The first section consists of an INPUT statement and an IF statement that tests whether the user pressed the INTERRUPT key to terminate the INPUT statement. The INPUT statement contains an AFTER FIELD clause that automatically calls the **statehelp** function after the user moves the cursor out of the State field on the Customer Form. (The **statehelp** function is described in the following section.) The user can terminate the INPUT statement by pressing either the ACCEPT key or the INTERRUPT key. If the user aborts the process by pressing the INTERRUPT key, INFORMIX-4GL sets the global variable **int_flag** to TRUE. The IF statement following the INPUT statement tests whether the INTERRUPT key has been pressed. If the value of **int_flag** is TRUE, INFORMIX-4GL resets the **int_flag** variable, displays a message, and leaves the **input_cust** function. (The return value FALSE indicates to the calling function that data entry has been aborted.)

If the user terminates the INPUT statement by pressing the ACCEPT key, INFORMIX-4GL executes the statements in the second section of the **input_cust** function. This section handles the insertion of data entered on the form into the database. RDSQL stores the value it automatically assigns to **customer_num** in the component SQLERRD[2] of the global record SQLCA (SQL Communications Area, a standard SQL language feature). **input_cust** retrieves this value and displays it on the screen in the Number field along with an appropriate message. The return value of TRUE indicates to the calling function that customer data has been entered into the database.

The **statehelp** Function

The **statehelp** function shown in Figure 13 tests whether the entry in the State field is valid. The function is called automatically after the user moves the cursor out of the State field.

```
FUNCTION statehelp()
    DEFINE idx INTEGER

    SELECT COUNT(*) INTO idx FROM state
        WHERE code = p_customer.state
    IF idx = 1 THEN
        RETURN
    END IF

    OPEN WINDOW w_state AT 8,40
        WITH FORM "state_list"
        ATTRIBUTE (BORDER, RED, FORM LINE 2)

    CALL set_count(state_cnt)
    DISPLAY ARRAY p_state TO s_state.*
    LET idx = arr_curr()

    CLOSE WINDOW w_state
    LET p_customer.state = p_state[idx].code
    DISPLAY BY NAME p_customer.state ATTRIBUTE (YELLOW)
END FUNCTION
```

Figure 13. The **statehelp** Function

The **statehelp** function uses a SELECT statement to test whether the current entry in the State field exists in a state table. If the entry exists, the function ends; otherwise, the function displays the state selection window shown in Figure 4 by performing the following operations:

1. Opens a window that displays a form containing the **s_state** screen array.
2. Displays the values in the **p_state** program array in the screen array. (The program array has already been filled with the names and abbreviations of the states.)

Figure 14 shows how the data scrolls automatically when the user moves the cursor with the arrow keys. The user can make a selection by moving the cursor to a state code and pressing the ACCEPT key.

Code	Name
AL	Alabama
AK	Alaska
[AZ]	[Arizona]
[AR]	[Arkansas]
[CA]	[California]
[CO]	[Colorado]
[CT]	[Connecticut]
[DE]	[Delaware]
[FL]	[Florida]
GA	Georgia
HI	Hawaii
...	

Figure 14. Automatic Scrolling of Data

3. Saves the position of the cursor in the **p_state** array.
4. Closes the state selection window.
5. Determines the state code from the position of the cursor in the **p_state** array.
6. Displays the state code in the State field of the original Customer Form (see Figure 5).

Query by Example

There are several places in the complete application where the user must select a particular customer. Since it is not known in advance what criteria the user might have for selection, the code provides a function **query_customer** that allows the user to select the customer by merely filling a form. With only a few lines of code, a program can allow the user to enter whatever data is known about the customer, display all customers satisfying the given criteria, and prompt the user to select one of the customers displayed. For the purpose of this overview, you can imagine that the code has displayed the form shown in Figure 15.

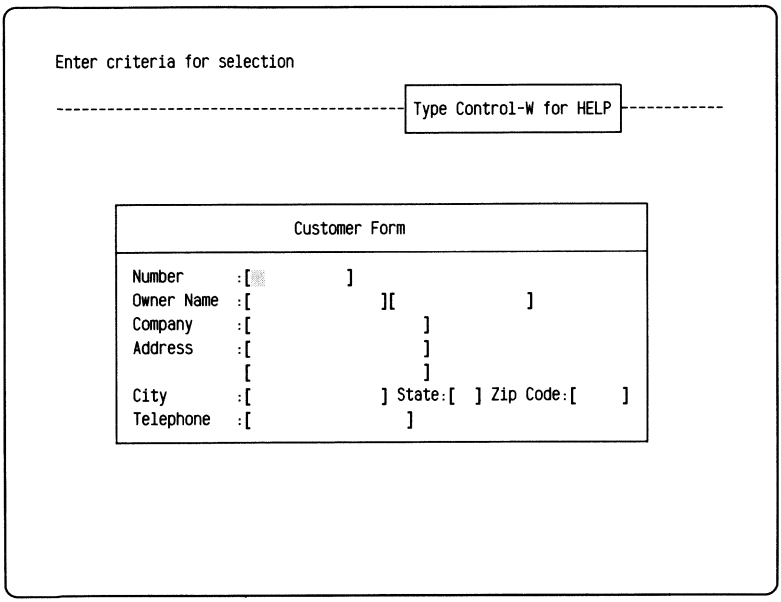

Figure 15. Query-by-Example Form

The **query_customer** function has four sections. The first section, shown in Figure 16, defines the variables that will be used, clears the form of any entries left over from previous activities, and displays a message on the second screen line instructing the user to enter query specifications.

```
FUNCTION query_customer()
    DEFINE where_text CHAR(200),
           query_text CHAR(250),
           answer     CHAR(1),
           chosen,
           exist      INTEGER
    CLEAR FORM
    CALL clear_menu()
    MESSAGE "Enter criteria for selection"
```

Figure 16. Section 1 of **query_customer**

The second section (Figure 17) turns the query-by-example input from the user into an executable RDSQL statement in four steps:

1. INFORMIX-4GL constructs a string **where_part** from the user input. For the input illustrated in Figure 6, the CONSTRUCT statement automatically assigns the following string to **where_part**:

   ```
   customer.customer_num>"115" and customer.lname matches "B*" and customer.city="Oakland"
   ```

2. The code creates a larger string by appending **where_part** to the end of the string "select * from customer where " and calls the result **query_text**. The CLIPPED function removes all trailing blanks.

3. The code associates the string query_text with the statement identifier **statement_1**.

4. The program names **customer_set** as the cursor (a pointer) to the current row, if any, that results from executing the SELECT statement.

```
CONSTRUCT where_part ON customer.* FROM customer.*
LET query_text = "select * from customer where ", where_part CLIPPED
PREPARE statement_1 FROM query_text
DECLARE customer_set CURSOR FOR statement_1
```

Figure 17. Section 2 of **query_customer**

The critical part of this program is the non-procedural CONSTRUCT statement. Like the INPUT statement, the CONSTRUCT statement allows the user to move from field to field and to enter all sorts of data, data ranges, pattern matches, and alternatives in each field. Only when the user presses the ACCEPT key does the program control pass to the next line of code.

The third section of the **query_customer** function (Figure 18) presents the user with the rows found by executing the query.

```
MESSAGE ""
LET chosen = FALSE
LET exist = FALSE
FOREACH customer_set INTO p_customer.*
    LET exist = TRUE
    DISPLAY BY NAME p_customer.*
    PROMPT "Press 'y' to select customer or RETURN to view next customer: "
        FOR CHAR answer
    IF answer MATCHES "[yY]" THEN
        LET chosen = TRUE
        EXIT FOREACH
    END IF
END FOREACH
```

Figure 18. Section 3 of **query_customer**

The third section begins by erasing the message line and setting two flags to FALSE. The flag **chosen** signals whether the user has chosen a row, while the flag **exist** signals whether any rows were found at all. The FOREACH statement opens the cursor **customer_set** and starts a loop that performs a series of fetches from the database, displaying the rows returned one at a time on the screen. The user is prompted to type 'y' to select the displayed row or to press [RETURN] to view the next row. If the user presses 'y' or 'Y', the flag **chosen** is set to TRUE and the program leaves the loop. Otherwise, the loop is repeated with the next row until no more rows are left.

The final section (Figure 19) of the **query_customer** function tests the flags, writes appropriate messages, and returns TRUE only if the user selects a row.

```
    IF NOT exist THEN
        MESSAGE "No customer satisfies query"
        LET p_customer.customer_num = NULL
        RETURN FALSE
    END IF
    IF NOT chosen THEN
        CLEAR FORM
        LET p_customer.customer_num = NULL
        MESSAGE "No selection made"
        RETURN FALSE
    END IF
    RETURN TRUE
END FUNCTION
```

Figure 19. Section 4 of **query_customer**

Reports

Getting information out of the database and formatting it for printing remains a central purpose for most database applications. One of the reports in the demonstration application creates mailing labels for selected customer rows that have been ordered by zip code. The corresponding program excerpts consist of two parts: the function **print_labels** (Figure 20) that is called from the REPORT Menu and selects the data, and the non-procedural report **labels_report** (Figure 21) that describes how the data should be formatted.

```
FUNCTION print_labels()
    DEFINE where_part   CHAR(200),
           query_text   CHAR(250),
           file_name    CHAR(20)

    DISPLAY FORM customer
    CALL clear_menu()
    DISPLAY "CUSTOMER LABELS:" AT 1,1
    MESSAGE "Use query-by-example to select customer list"

    CONSTRUCT where_part ON customer.* FROM customer.*
    LET query_text = "select * from customer where ", where_part CLIPPED,
                     " order by zipcode"
    PREPARE statement_1 FROM query_text
    DECLARE label_list CURSOR FOR statement_1

    CLEAR SCREEN
    PROMPT "Enter file name for labels )" FOR file_name
    CLEAR SCREEN
    MESSAGE "Printing mailing labels to ", file_name CLIPPED, " -- Please wait"

    START REPORT labels_report TO file_name
    FOREACH label_list INTO p_customer.*
        OUTPUT TO REPORT labels_report (p_customer.*)
    END FOREACH
    FINISH REPORT labels_report

    MESSAGE "Labels printed to ", file_name CLIPPED
END FUNCTION
```

Figure 20. A Sophisticated Program Excerpt That Calls a Report

In the same way that **query_customer** used a query by example to select a set of customers, **print_labels** selects the set of customers for which it will print labels. The differences here are that the SELECT statement has an ORDER BY clause that causes the output to be sorted by zip code and the cursor is named **label_list.**

After the query-by-example processing, the program prompts the user to enter the name of a file to contain the labels. It then displays a message that the labels are being printed. The report writing is governed by the next five lines and consists of three steps. The START REPORT statement initiates the report writing process and designates the file to contain the labels. The next stage is a FOREACH loop that takes one row from the SELECT statement at a time and delivers it to the report. The last step is the FINISH REPORT statement that handles any end-of-report processing.

```
REPORT labels_report (r)
    DEFINE r RECORD LIKE customer.*

    OUTPUT
        TOP MARGIN 0
        BOTTOM MARGIN 0
        PAGE LENGTH 6

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

Figure 21. Program Excerpt Describing a Report

The report routine is named **labels_report.** There are two sections to this routine: the OUTPUT section and the FORMAT section. The OUTPUT section describes some output parameters: the margins and the page length (given here as six lines; each label will occupy a "full page").

The FORMAT section describes how the report should be organized on the page. For each row that is handed to the report, the report will skip to the top of the next page as defined in the OUTPUT section. The first row of each label contains the first name (with following blanks clipped off), a space, and the last name of the customer. The second row of the label is the customer's company and the third is the first address line. If a second address line exists, it will be the fourth line. The last line contains, in usual manner for labels, the city, a comma and a space, the state, two spaces, and the zip code.

This is a particularly simple report and only hints at the power of the report-writing capability of INFORMIX-4GL. You can write reports with special first-page formatting, headers on subsequent pages, and aggregate values such as percentages, sums, averages, maximums, and minimums-not only for the entire report but for groups of rows within the report. Your reports can be sent directly to the printer. For example, the demonstration application automatically writes an invoice when you enter a new order. Figure 22 shows the partial output of another report that includes page headers, subtotals, and totals and that requires only a half page of code.

West Coast Wholesalers, Inc.
Statement of ACCOUNTS RECEIVABLE - Jul 12, 1986

Ludwig Pauli/All Sports Supplies

Order Date	Order Number	Ship Date	Amount
06/01/1986	1002	06/06/1986	$1,200.00
			$1,200.00

West Coast Wholesalers, Inc.
Statement of ACCOUNTS RECEIVABLE - Jul 12, 1986

Anthony Higgins/Play Ball!

Order Date	Order Number	Ship Date	Amount
01/20/1986	1001	02/01/1986	$250.00
03/23/1986	1011	04/13/1986	$99.00
09/01/1986	1013	09/13/1986	$143.80
10/12/1986	1003	10/13/1986	$959.00
			$1,451.80

West Coast Wholesalers, Inc.
Statement of ACCOUNTS RECEIVABLE - Jul 12, 1986

George Watson/Watson & Son

Order Date	Order Number	Ship Date	Amount
04/12/1986	1004	04/30/1986	$2,126.00
05/01/1986	1014	05/10/1986	$1,440.00
			$3,566.00

Figure 22. Partial Accounts Receivable Report

28

These brief excerpts give you the flavor of writing an application using INFORMIX-4GL. They use uppercase letters for keywords so that you can distinguish the keywords of the language from the programmer-invented identifiers. INFORMIX-4GL is case insensitive and ignores these distinctions.

INFORMIX-4GL represents a creative solution to the dilemma caused by the unavoidable tension between flexibility and simplicity. Its basic statements are simple; its optional extensions are rich. You can write useful programs with very few lines of code. We designed INFORMIX-4GL to be a complete data processing language. It is extremely unlikely that you will find things that you cannot do within the options of the syntax. An interface to the C programming language exists if you need it (for returning a cosine, for example).

In the examples that have been presented, you may have seen more procedural steps in the INFORMIX-4GL program excerpts than you think a fourth-generation language should have. There are two reasons for this.

The first reason is that the non-procedural parts are so terse and so powerful that they take fewer lines of code to write. The MENU, INPUT, CONSTRUCT, and window statements (and a number of other statements not illustrated here) are very compact and yet they handle the lion's share of the application. The procedural statements do the bookkeeping and are much less succinct. INFORMIX-4GL programs do have a fair amount of procedural statements, but only because the non-procedural ones do so much work.

The second reason for having procedural statements is that the procedural statements enable you to do things that the designers of INFORMIX-4GL could not predict. You do not have to use them all, but the chances are that when you need a special effect to implement your application, INFORMIX-4GL will meet your need by providing you with the proper tools.

If you have written customized database applications using a third-generation language, you will find that switching to INFORMIX-4GL turns a difficult and error-filled process into an enjoyable one. If you have not customized your application because you did not have the necessary programming skills, the ease of programming with INFORMIX-4GL will dramatically expand the power you have over your database. A new and simple tool for creating sophisticated database applications is now available. The distance between application planning and implementation has just become much shorter for everyone.

Recap of INFORMIX-4GL

INFORMIX-4GL joins five other SQL products:

- INFORMIX-SQL
- INFORMIX-ESQL/C (Embedded SQL and Tools for C)
- INFORMIX-ESQL/COBOL (Embedded SQL for COBOL)
- *File-it!*
- C-ISAM

We designed these products to meet the database management needs of people with a wide spectrum of computer sophistication. INFORMIX-4GL satisfies the application-building needs of the widest audience.

INFORMIX-SQL

INFORMIX-SQL is the relational database management system that developers choose to create custom applications. Based on RDSQL, INFORMIX-SQL has the tools to create and maintain databases, design custom screens and menus, and produce custom-formatted reports. It also provides an interactive, application-building environment that permits interactive manipulation of a database using the SQL language. Through the menu-creating features of INFORMIX-4GL, your 4GL programs can invoke applications built with INFORMIX-SQL and make them part of a broader application. INFORMIX-SQL can enhance your programming with INFORMIX-4GL by providing you the ability to test your database queries and operations from the Programmer's Environment through the use of the interactive query language.

INFORMIX Embedded SQL and Tools for C

This package offers two sets of complementary tools for highly specialized applications. First, INFORMIX-ESQL/C makes it easy for you to embed RDSQL statements into your C code. With its advanced capabilities, you can prepare dynamic queries and manipulate databases using SQL without leaving your C program. A companion product, INFORMIX-ESQL/COBOL, makes these same features (with the exception of handling dynamic queries) available to COBOL programmers. Second, INFORMIX-ESQL/C includes programming

Other SQL Products

tools that make it possible for you to call C functions and special C library routines while working in Ace (the INFORMIX-SQL report writer) and Perform (the INFORMIX-SQL form transaction processor). Although you can perform virtually every sophisticated database activity solely with INFORMIX-4GL, you also have the option to call functions created with INFORMIX-ESQL/C in your 4GL programs.

File-it!
File-it! is a completely menu-driven, interactive file manager that makes it simple to create and maintain tables in a database, to make screen queries, and to print reports based on a single table. You can use the tables created by *File-it!* with the other Informix products, and vice versa.

C-ISAM
C-ISAM is the foundation of Informix data management. It consists of a library of C-language functions for creating and managing indexed file systems. As the standard access method for the UNIX operating system, C-ISAM performs all the required index file maintenance and manipulation tasks and provides fast data access and retrieval, efficient storage, and comprehensive protection. All SQL-based products use the C-ISAM file structure for their data and index files.