HCL Informix 4GL Interactive Debugger

Version 7.51

Disclaimer: *Informix is a trademark of IBM Corporation in at least one jurisdiction and is used under license.*

Contents

Introduction

In This Introduction

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

About This Manual

This manual is both an introduction to the INFORMIX-4GL Interactive Debugger and a comprehensive reference of Debugger commands and features. The organization of the guide is summarized here and discussed in detail in the next section:

- Chapter 1 presents an overview of Debugger features, while Chapters 2 through 7 introduce the debugging environment and present typical debugging strategies in a series of debugging sessions. As you work through these sessions, you will learn debugging strategies suited to both single- and multi-module 4GL application programs.
- Chapters 8 and 9 are reference chapters that provide a detailed discussion of all Debugger commands and features.

Organization of This Manual

To meet the needs of both new and experienced Debugger users, this guide is in two parts: a seven-chapter tutorial and a two-chapter reference section.

The tutorial chapters are as follows:

- Chapter 1, "Introduction to the Debugger," presents an overview of Debugger features.
- Chapter 2, "Getting Started with the Debugger," introduces the debugging environment and discusses basic operations such as how to invoke the Debugger from the Programmer's Environment and how to exit from the Debugger.
- Chapter 3, "Tracing Logic of the customer Program," presents a debugging session with the single-module **customer** program (supplied with the Debugger software). To familiarize yourself with the **customer** program logic, you will use *tracepoints*, an important Debugger feature.
- Chapter 4, "Analyzing a Logical Error in the customer Program," continues debugging of the **customer** program. As you diagnose a logical program error, you will learn how *breakpoints* suspend program execution so you can gather information or interact with the program.
- Chapter 5, "A Multi-Module Program: cust_order," contains important information about multi-module programs, such as how to compile them from the Programmer's Environment, how to display specified modules in the Source window, and how to set breakpoints and tracepoints within selected modules.
- Chapter 6, "Tracing Logic of the cust_order Program," adds to your knowledge of breakpoints and tracepoints as debugging of the **cust_order** program continues. The chapter introduces the CALL command to interactively call a function.
- Chapter 7, "Analyzing Runtime Errors in the cust_order Program," concludes debugging of the **cust_order** program as you learn how to diagnose runtime errors.

The reference chapters are as follows:

- Chapter 8, "The Debugging Environment," discusses the debugging environment in detail and explains how you can customize the environment to suit your needs.
- Chapter 9, "The Debugger Commands," is a detailed, alphabetical listing of Debugger command syntax.

The appendixes are as follows:

- Appendix A, "Environment Variables," describes the environment variables recognized by 4GL and the Debugger.
- Appendix B, "Calling C Functions," illustrates how to use the Debugger to analyze 4GL programs that call programmer-defined C functions or INFORMIX-ESQL/C functions.
- Appendix C, "Sample Programs," discusses the sample programs used during the debugging sessions in this guide.
- The "Error Messages" section lists the Debugger error codes, explains their meanings, and suggests ways to avoid the errors.

Types of Users

This manual is written for all 4GL users. You do not need database management experience or familiarity with relational database concepts to use this manual. Knowledge of SQL (Structured Query Language) and experience using a high-level programming language are useful.

To follow the logic of the sample debugging sessions, you need a general understanding of 4GL statement syntax and features. You should also be familiar with the Programmer's Environment, specifically, how to correct and recompile 4GL program modules. (You will learn how to compile multi-module programs in the tutorial chapters.)

Software Dependencies

This manual is written with the assumption that you are using an Informix database server, Version 14.10 or later.

Assumptions About Your Locale

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

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

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

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

Documentation Conventions

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

The following conventions are discussed:

- Typographical conventions
- Icon conventions
- Syntax conventions

Typographical Conventions

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

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

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

Icon Conventions

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

Feature, Product, and Platform Icons

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

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

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

Compliance Icons

Compliance icons indicate paragraphs that provide guidelines for complying with a standard.

Icon	Description
ANSI	Identifies information that is specific to an ANSI-compliant database
X/O	Identifies functionality that conforms to X/Open
+	Identifies information that is an Informix extension to ANSI SQL-92 entry-level standard SQL

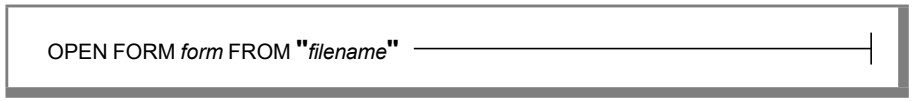
These icons can apply to a row in a table, one or more paragraphs, or an entire section. A ♦ symbol indicates the end of the compliance information.

Syntax Conventions

SQL statement syntax is described in the *Informix Guide to SQL: Syntax*. The syntax of other 4GL statements is described in the *INFORMIX-4GL Reference*.

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

Figure 1
Example of a Simple Syntax Diagram

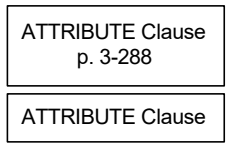
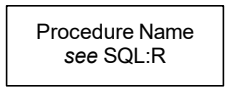


Each syntax diagram begins at the upper-left corner and ends at the upper-right corner with a vertical terminator. Between these points, any path that does not stop or reverse direction describes a possible form of the statement. (For a few diagrams, notes in the text identify path segments that are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

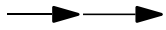
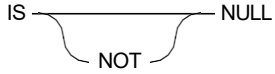
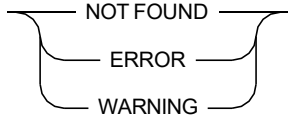
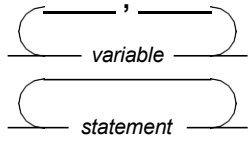
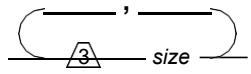
Syntax elements in a path represent terms, keywords, symbols, and segments that can appear in your statement. The path always approaches elements from the left and continues to the right, except in the case of separators in loops. For separators in loops, the path approaches counterclockwise. Unless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

Elements That Can Appear on the Path

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

Element	Description
KEYWORD	A word in UPPERCASE letters is a keyword. You must spell the word exactly as shown; however, you can use either uppercase or lowercase letters.
(. , ; @ + * - /)	Punctuation and other nonalphanumeric characters are literal symbols that you must enter exactly as shown.
" " ' '	Double quotes must be entered as shown. If you prefer, you can replace the pair of double quotes with a pair of single quotes, but you cannot mix double and single quotes.
variable	A word in italics represents a value that you must supply. A table immediately following the diagram explains the value.
	A reference in a box represents a subdiagram. Imagine that the subdiagram is spliced into the main diagram at this point. When a page number is not specified, the subdiagram appears on the same page. The aspect ratios of boxes are not significant.
	A reference to SQL:R in a syntax diagram represents an SQL statement or segment that is described in the *Informix Guide to SQL: Reference*. Imagine that the segment were spliced into the diagram at this point.

(1 of 2)

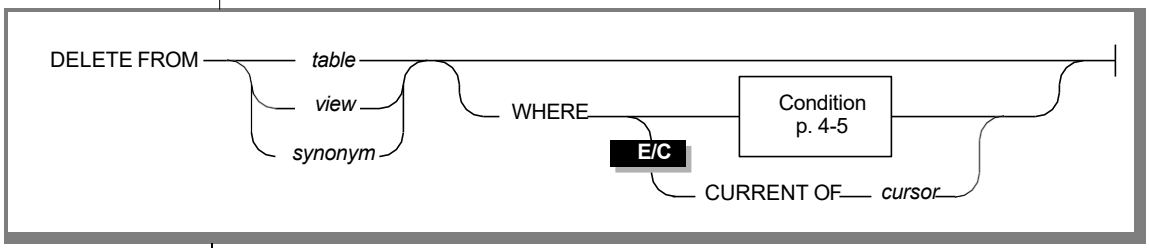
Element	Description
SE	An icon is a warning that this path is valid only for some products, or only under certain conditions. Characters on the icons indicate what products or conditions support the path. These icons might appear in a syntax diagram: **SE** This path is valid only for INFORMIX-SE. **IDS** This path is valid only for Informix Dynamic Server.
ALL	A shaded option is the default action.
	Syntax within a pair of arrows is a subdiagram.
	The vertical line terminates the syntax diagram.
	A branch below the main path indicates an optional path. (Any term on the main path is required, unless a branch can circumvent it.)
	A set of multiple branches indicates that a choice among more than two different paths is available.
	A loop indicates a path that you can repeat. Punctuation along the top of the loop indicates the separator symbol for list items. If no symbol appears, a blank space is the separator.
	A gate (3) on a path indicates that you can only use that path the indicated number of times, even if it is part of a larger loop. You can specify *size* no more than three times within this statement segment.

(2 of 2)

How to Read a Syntax Diagram

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

Figure 2
Example of a Syntax Diagram



To use this diagram to construct a statement, start at the top left with the keyword DELETE FROM. Then follow the diagram to the right, proceeding through the options that you want.

Figure 2 illustrates the following steps:

1. Type `DELETE FROM`.
2. You can delete a table, view, or synonym:
   - Type the table name, view name, or synonym, as you desire.
   - You can type `WHERE` to limit the rows to delete.
   - If you type `WHERE` and you are using DB-Access or the SQL Editor, you must include the Condition clause to specify a condition to delete. To find the syntax for specifying a condition, go to the "Condition" segment on the specified page.
   - If you are using ESQL/C, you can include either the Condition clause to delete a specific condition or the CURRENT OF *cursor* clause to delete a row from the table.
3. Follow the diagram to the terminator.

   Your DELETE statement is complete.

Chapter

1 Introduction to the Debugger

In This Chapter

This chapter introduces important concepts about the INFORMIX-4GL Interactive Debugger:

- What a debugger is
- How you can use the Debugger to learn more about programs created with the INFORMIX-4GL Rapid Development System
- What the principal commands and capabilities of the Debugger are

  Step-by-step instructions for using the Debugger begin in Chapter 2, "Getting Started with the Debugger."

Introducing the Debugger

The Debugger is a set of tools that allow you to interact with your INFORMIX-4GL programs while they are running. You can perform the following tasks with the Debugger:

- Quickly familiarize yourself with programs or program segments that someone else has written.
- Determine the source of errors within your programs.
- Learn more about the workings of the 4GL language.

If you suspect, for example, that your program is not producing correct output, and you do not have the Debugger, you can take some or all of the following actions:

- Review your source code.
- Trace the execution of the program on paper, using sample data. This is sometimes called *desk checking*.
- Place additional DISPLAY statements in your program to print the contents of variables at various points of program execution.

The Debugger makes it easier to verify that your 4GL programs work correctly, and allows you to detect and fix errors much more efficiently than with these conventional techniques. You can use the Debugger to discover the cause of both *logical* errors (errors that cause the program to produce undesirable results) and *fatal* errors (errors that prevent the program from continuing execution).

Specifically, you can take the following actions with the Debugger:

- Interrupt a program and resume execution from the point of interruption.
- Review your source code as it executes, stepping through as many lines at a time as you desire.
- Change the values of program variables, and resume execution with the new values.
- Set *tracepoints* to monitor the execution of a specific line of code or function, or when the value of a variable changes.
- Set *breakpoints* to suspend program execution at a specific line of code or function, when the value of a variable changes, or when a specific condition becomes true.

The Debugger is a source language debugger. You do not need to know any programming language other than 4GL in order to use the Debugger. You can access the Debugger either from the 4GL Programmer's Environment or from the command line.

Using the Debugger

The following sections describe the major features and capabilities of the Debugger, and illustrate ways in which you can use the Debugger to learn more about your 4GL programs.

The Debugger Screens

To facilitate interaction with your programs, the Debugger divides the terminal environment into two screens. These are as follows:

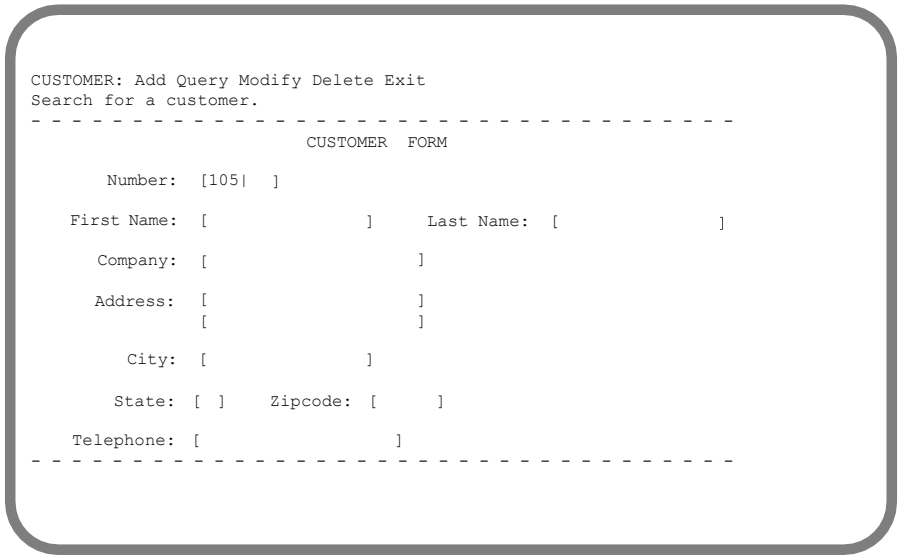
- The Application screen
- The Debugger screen

The Application Screen

The Application screen is used to display the input and output of your 4GL program. The program appears exactly as if it were running outside the Debugger. You can make menu selections, enter and retrieve data, and respond to prompts as usual.

For example, Figure 1-1 illustrates the appearance of the Application screen during a debugging session with the **customer** program introduced in Chapter 2, "Getting Started with the Debugger." The **Query** option has been chosen, and search criteria are being entered into the form.

Figure 1-1
The Application Screen with the customer Program

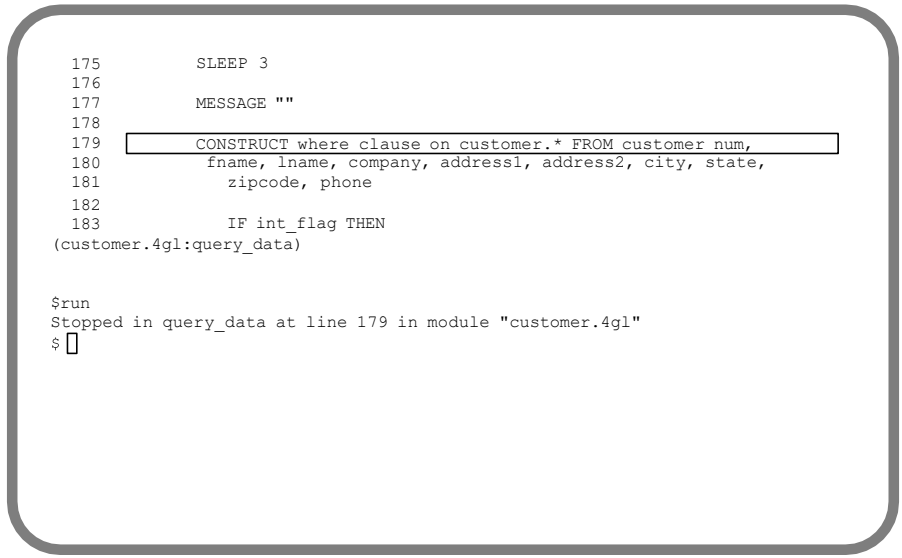


The Debugger Screen

To monitor source code, and to interact with an executing program, you must suspend execution of the program and display the Debugger screen. You can display the Debugger screen any time a program is running by pressing the **Interrupt** key (usually DEL or CTRL-C).

If you now press the **Interrupt** key to suspend execution of the **customer** program, the Debugger screen appears as in Figure 1-2.

Figure 1-2
The Debugger Screen with the customer Program

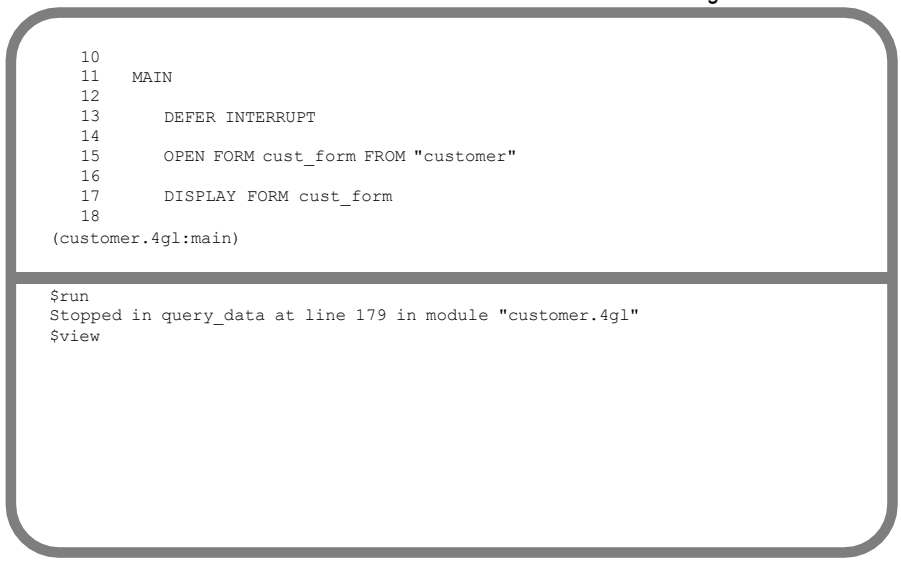


The Source Window

The Debugger uses two separate portions of the Debugger screen, or *windows*, for its activities. The top, or Source, window is used for displaying source code. When execution is suspended, the Source window displays the currently executing program segment. A highlighting bar indicates the next segment to execute when you resume execution.

The source module can be scrolled through this window, using certain CTRL keys or the UP ARROW and DOWN ARROW keys. For example, you can scroll through the program until the MAIN section is displayed, as in Figure 1-3.

Figure 1-3
Scrolling the Source Module

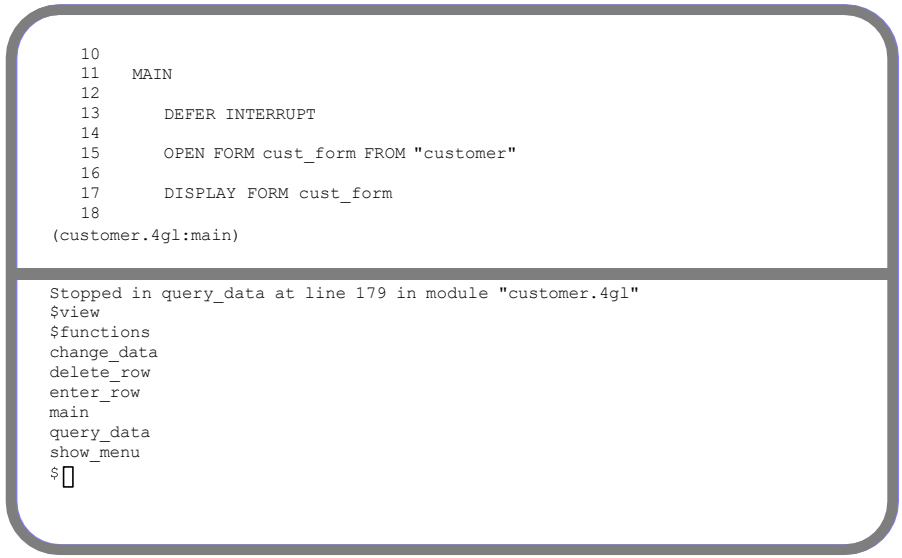


The Command Window

The bottom, or Command, window is used for entering Debugger commands. Output from these commands is also displayed here. When you first access the Debugger, or when you suspend execution of a program, the cursor appears at the $ prompt in this window, and the Debugger awaits input.

For example, if you enter a FUNCTIONS command, the Debugger lists all the programmer-defined functions in the program in the Command window, as shown in Figure 1-4. You can scroll the contents of the Command window using the same keys that you use to scroll the Source window.

Figure 1-4
Output of the FUNCTIONS Command to the Command Window



The Debugger provides a number of parameters that allow you to easily control the interaction of the Debugger and Application screens. For example, you can tell the Debugger to highlight each statement in the Source window as it executes. You can also specify whether the Application screen should appear whenever your program produces output, or only when it requires input.

On a single terminal, you can view either the Application screen or the Debugger screen at one time. If you have two terminals that use the same **termcap** or **terminfo** entry, you can use a separate terminal for each screen. (This feature is discussed in greater detail in Chapter 8, "The Debugging Environment," and Chapter 9, "The Debugger Commands.")

Tracepoints

By setting *tracepoints*, you can monitor when a particular function or statement executes, or when the value of a variable changes. Tracepoints are useful tools for tracing the flow of control of an unfamiliar program. They are also a valuable debugging device if, for example, you suspect that a function is not returning the correct values.

The example in Figure 1-5 illustrates the output of a tracepoint that has been set to monitor the execution of the **get_stock** function in the **cust_order** program introduced in Chapter 5, "A Multi-Module Program: cust_order." This tracepoint was set with the following Debugger command:

```
trace  get_stock
```

Figure 1-5
Output of a Tracepoint in the cust_order Program

```
   21          CLEAR FORM
   22          ERROR "Order input aborted" ATTRIBUTE (RED, REVERSE)
   23          RETURN
   24       END IF
   25       INPUT ARRAY p_items FROM s_items.*
   26          BEFORE FIELD stock_num
   27          MESSAGE "Press ESC to write order"
   28          DISPLAY "Enter a stock number or press CTRL-B to
                 scan stock list"

(order.4gl:add_order)
```

```
$trace get_stock
(1)  trace in function get_stock [order.4gl]
          scope function: get_stock
$run
Enter get_stock() from add_order line 45
Return (2, "HRO", "baseball       ", $126.00) from get_stock at line 195
```

The tracepoint outputs the line number where the call to **get_stock** was made, along with the name of the calling function. When **get_stock** ends, the tracepoint outputs the line number at which execution terminated. It also displays the values that **get_stock** returns to the calling function.

Breakpoints

Breakpoints allow you to suspend program execution when a preset condition occurs. These conditions include when a particular statement or function executes, or when the value of a variable changes. You can also specify an IF condition.

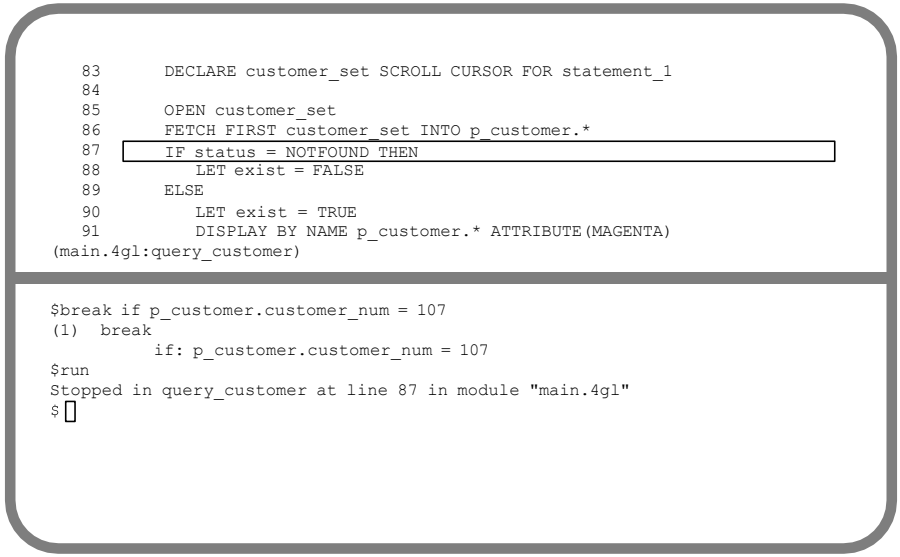
While execution is suspended, you can perform activities such as viewing the current values of variables, or listing the functions that were called to arrive at the current statement. You can change the value of a variable, and resume execution with the new value. You can set breakpoints at program segments that you want to examine in detail, and execute your program one statement at a time after the breakpoint is reached.

Program execution does not resume following a breakpoint until you issue a specific Debugger command to do so.

In Figure 1-6 on page 1-12, a breakpoint causes execution to be suspended in the **cust_order** program if the value of the global record member **p_customer.customer_num** is `107`. This breakpoint was set with the following Debugger command:

```
break  IF  p_customer.customer_num  =  107
```

Figure 1-6
Output of a Breakpoint in the cust_order Program

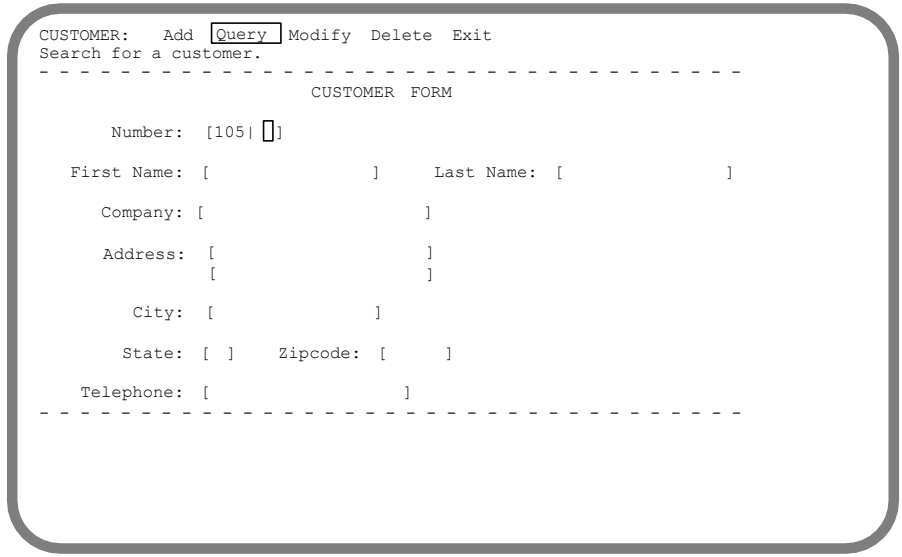


Resuming Execution of a Program

An indispensable feature of a debugger is the ability to resume operation of a program from a point of interruption. Interruption might have occurred because a breakpoint was reached, or because you pressed the **Interrupt** key (typically CTRL-C).

The CONTINUE command resumes execution of the program from the point of interruption. When you use CONTINUE, there is no further interruption of the program unless a breakpoint is reached, or another **Interrupt** is entered. For example, if you interrupted the **customer** program during execution of the CONSTRUCT statement, as shown in Figure 1-1 on page 1-6, you can now use the CONTINUE command to resume operation. The Application screen reappears with the query still in progress, as shown in Figure 1-7.

Figure 1-7
Resuming Execution with CONTINUE



As an alternative to the CONTINUE command, you can use the STEP command to execute your statements individually, or by as many lines as you choose. Stepping through 4GL statements is an excellent way to familiarize yourself with the flow of control of unfamiliar programs. When you use the STEP command, the Debugger highlights the next statement to execute in the Source window. It displays in the Command window the line number and function of the statement just executed.

In Figure 1-8, the STEP command executes the next statement in the **cust_order** program following a breakpoint.

Figure 1-8
Stepping Through Statements

```
   83       DECLARE customer_set SCROLL CURSOR FOR statement_1
   84
   85       OPEN customer_set
   86       FETCH FIRST customer_set INTO p_customer.*
   87       IF status = NOTFOUND THEN
   88          LET exist = FALSE
   89       ELSE
   90          LET exist = TRUE
   91          DISPLAY BY NAME p_customer.* ATTRIBUTE(MAGENTA)
(main.4gl:query_customer)

$break if p_customer.customer_num = 107
(1)  break
          if: p_customer.customer_num = 107
$run
Stopped in query_customer at line 87 in module "main.4gl"
$step
Stopped in query_customer at line 90 in module "main.4gl"
$
```

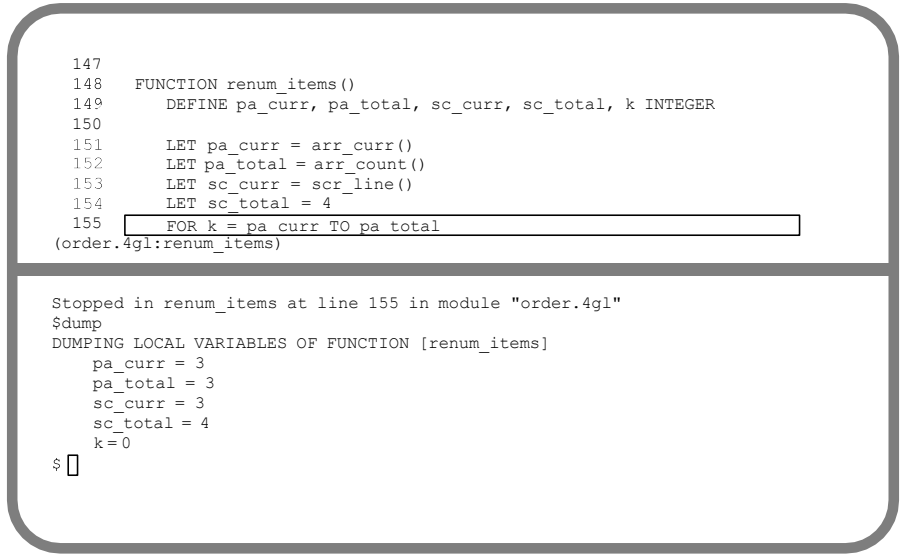
Viewing the Values of Program Variables

The Debugger provides two simple and convenient commands for viewing the values of program variables. Without the Debugger, it would be necessary to place individual DISPLAY statements in your code to monitor the values of variables.

The DUMP command allows you to display the values of local, global, and module variables in the currently executing function. If you specify the GLOBALS option, the DUMP command lists the values of all programmer-defined global or module variables as well as the values of the 4GL global variables such as **status** and the members of the SQLCA record.

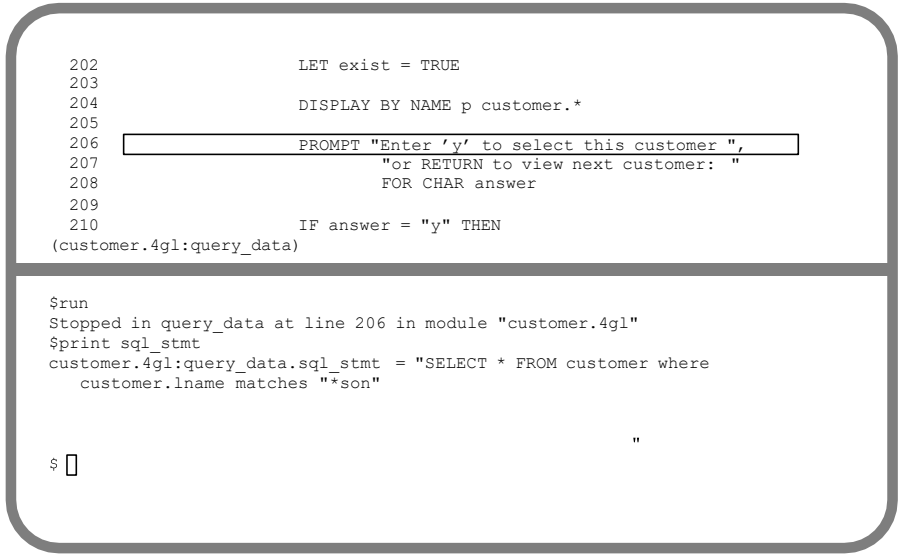
Figure 1-9 illustrates the use of the DUMP command to display the current values of all local variables during execution of the **renum_items** function in the **cust_order** program.

Figure 1-9
The DUMP Command



The PRINT command allows you to display the value of an individual variable in an active function. With this single command, you can print the value of a simple variable, the values of all the members of a program record, or the values of all the elements of a program array. The following two examples illustrate sample output of the PRINT command with examples from the **customer** program. In Figure 1-10 on page 1-16, the PRINT command displays the value of the character string **sql_stmt** in the Command window.

Figure 1-10
Printing the Value of a Variable



In the next example, the PRINT command has been used to save the values of the members of the **p_customer** program record in a file. The name of the file is **print1**, and the values assigned to the record are those of Frank Albertson. The following command produced this file:

```
print p_customer >> print1
    global:p_customer   = record
        customer_num  =  114
        fname  = "Frank            "
        lname = "Albertson         "
        company = "Sporting Place          "
        address1 = "947 Waverly Place      "
        address2 = (null)
        city = "Redwood City      "
        state = "CA"
        zipcode  =  "94062"
        phone  =  "415-886-6677        "
    end  record
```

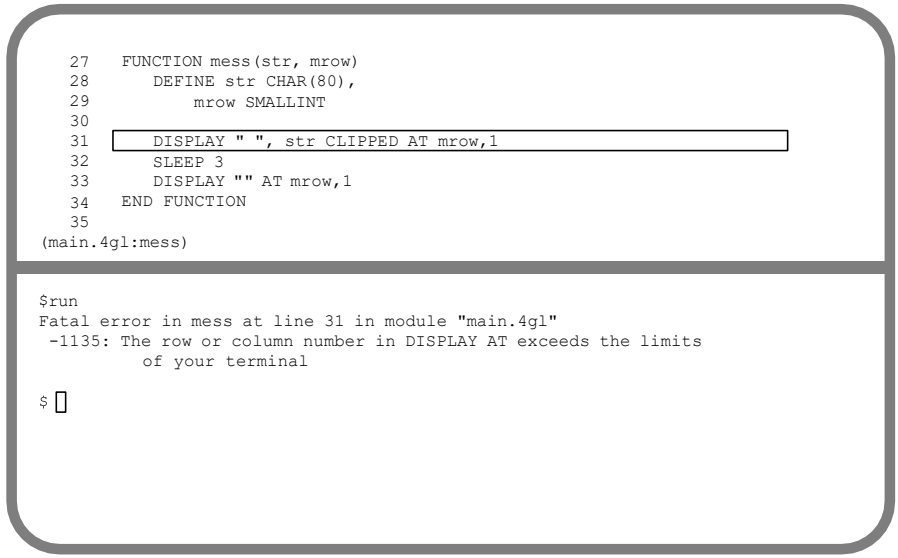
You can use both the DUMP and PRINT commands to display the values of variables automatically as tracepoints and breakpoints execute.

Analyzing Fatal Errors

The Debugger makes the task of diagnosing fatal errors much easier by allowing you to work with your program even after it has aborted. For example, you can list the functions that were called leading up to the statement that caused the program to terminate abnormally. You can also print the values of the program variables at the time execution terminated.

When a fatal error occurs, the Debugger immediately redisplays the Source and Command windows. The Source window highlights the statement at which execution terminated. The Command window tells you where in the program the error occurred, and displays the error number and message. Figure 1-11 illustrates the appearance of the Debugger screen after a fatal error has occurred in the **cust_order** program.

Figure 1-11
Encountering a Fatal Error



Following a fatal error, you can rerun the program from within the Debugger. In many instances, you can go right to the program segments you want to examine by recalling a specific function. If the function you want to call requires an active database, you can use the DATABASE command to reopen the database for this purpose.

Short Forms of Commands

For ease of entry, you can enter a Debugger command using the fewest number of characters that uniquely identify it. For example, you can enter the DUMP command as **du**.

You do not need to memorize the unique abbreviation for each Debugger command, and you can always enter additional characters. If you enter a sequence of characters that identify more than one command, the Debugger prompts you with the available choices.

For example, if you enter `r`, which can identify either the RUN or READ command, the Debugger responds as shown in Figure 1-12.

Figure 1-12
Short Forms of Commands

Function Keys

The most common Debugger commands have corresponding function keys F1 through F9. These keys are:

```
alias f1 = help
alias f2 = step
alias f3 = step into
alias f4 = continue
alias f5 = run
alias f6 = list break trace
alias f7 = list
alias f8 = dump
alias f9 = exit
```

Actions that you have assigned to function keys in your 4GL program (or used terminal setup to define) override the actions assigned to those keys by the Debugger while the application program is running. If desired, you can redefine the Debugger function keys to specify any other command or command sequence. (This feature is discussed in detail in Chapter 9.)

Requesting Help

Help on all the Debugger commands is always available. You can enter the HELP command, or press F1, to see the list of valid commands shown in Figure 1-13 on page 1-20.

Figure 1-13
Requesting Help

```
CHOOSE COMMAND TO DETAIL>>
Choose a command with Arrow keys, or enter a name, then press Return.

_escape             call              grow              step
_interrupt          cleanup           help              timedelay
_redraw             continue          let               trace
_screen             database          list              turn
_search             disable           nobreak           use
_toggle             dump              notrace           variable
alias               enable            print             view
application         exit              read              where
break               functions         run               write
```

You can choose a command from this list by highlighting it and pressing RETURN. For example, if you choose the VARIABLE keyword, the screen shown in Figure 1-14 appears.

Figure 1-14
The HELP Screen for the VARIABLE Command

```
HELP: [] [SCREEN] RESUME
Displays the next page of Help text.

------------------------------------------------------------------------------

  VARIABLE

    Overview

    Use VARIABLE to display the declaration of a program variable, or
      to save it in a file.

    Syntax

    ------------------------------------------------------------------------
    VARIABLE [variable | GLOBALS | ALL] [>>filename]
    ------------------------------------------------------------------------

    Explanation

    VARIABLE      is a required keyword.

    variable      is the name of a program variable.

    GLOBALS       is an optional keyword.
```

You can also request help on a particular Debugger command directly at the $ prompt. For example, the following command also displays the Help screen for the VARIABLE command:

```
help variable
```

The Debugger provides tools that are useful to 4GL programmers of every level of proficiency. If you are just starting to program with 4GL, you will find that the Debugger commands can be used to gain valuable insights into the workings of your programs and of the 4GL language. If you are an experienced programmer, you will find that the options provided for the basic commands are comprehensive enough to handle virtually any debugging task. Whatever your reasons for using a debugger, you will find that the Debugger is an invaluable companion to 4GL.

Chapter

2

Getting Started with the Debugger

In This Chapter

This chapter introduces the debugging environment. It shows you how to operate the INFORMIX-4GL Interactive Debugger and manipulate the Debugger screens using a single module program, **customer.4gl**. This and subsequent chapters assume that you are familiar with basic INFORMIX-4GL statement syntax, including the INPUT and CONSTRUCT statements, and with the SQL commands to INSERT, UPDATE, and DELETE database rows. The following topics are covered:

- How to access the Debugger
- How to manipulate the Source and Command windows
- How to use the LIST command to display the current debugging environment
- How to run the Debugger
- How to save the current debugging environment in a file
- How to exit from the Debugger

The examples in this chapter are based on the following program, form, and help files provided with the demonstration database:

```
customer.4gl
customer.per
custhelp.ex
```

For information about creating the demonstration database, see "Debugger Demonstration Database and Examples" on page 7.

To run the Debugger, your operating environment must be properly configured. Appendix A explains the environment variables that control the environment, including the **DBSRC** environment variable, which is recognized by the Debugger but not by 4GL. This variable can be used to augment your directory search path during a debugging session. You do not, however, need to specify **DBSRC** in order to use the Debugger.

The customer.4gl Program

The **customer.4gl** program is a single module program that allows the user to carry out the following activities:

- Add customer rows
- Query the **customer** table
- Modify a customer row returned by the query function
- Delete a customer row returned by the query function

A complete discussion of the **customer.4gl** program is provided in Appendix C, "Sample Programs." You should consult Appendix C if you would like more information on the program after studying the example.

An intentional bug has been coded into the program that produces undesirable results under certain conditions. In Chapter 3, "Tracing Logic of the customer Program," you use the Debugger to trace the logic of the **customer** program, and the way in which the values of program variables are shared among the various functions. Chapter 4, "Analyzing a Logical Error in the customer Program," illustrates the use of the Debugger to discover the logical error, and provides instructions for correcting the program.

A listing of the program and brief explanatory notes follow:

```
 1DATABASE stores7
 2
 3GLOBALS
 4
 5    DEFINE
 6        p_customer RECORD LIKE customer.*,
 7        chosen SMALLINT
 8
 9END GLOBALS
10
11MAIN
12
13    DEFER INTERRUPT
14
15    OPEN FORM cust_form FROM "customer"
16
17    DISPLAY FORM cust_form
18
19    LET chosen = FALSE
20
21    OPTIONS MESSAGE LINE 22,
22        PROMPT LINE 21,
23        HELP FILE "custhelp.ex",
24        HELP KEY CONTROL-I
```

```
25
26    CALL show_menu()
27
28    MESSAGE "End program."
29
30    SLEEP 3
31
32    CLEAR SCREEN
33
34END MAIN
35
36
37FUNCTION show_menu()
38
39    DEFINE answer  CHAR(1)
40
41    MESSAGE "Type the first letter of the option ",
42        "you want to select or CONTROL I for Help."
43
44    MENU "CUSTOMER"
45
46        COMMAND "Add" "Add a new customer." HELP 1
47
48            LET answer = "y"
49
50            WHILE answer = "y"
51
52                CALL enter_row()
53
54                PROMPT "Do you want to ",
55                   "enter another row (y/n) ? "
56                   FOR CHAR answer
57
58            END WHILE
59
60            CLEAR FORM
61
62        COMMAND "Query" "Search for a customer." HELP 2
63
64            CALL query_data()
65
66            IF chosen THEN
67
68                NEXT OPTION "Modify"
69
70            END IF
71
72        COMMAND "Modify" "Modify a customer." HELP 3
73
74            IF chosen THEN
75
76                CALL change_data()
77
78            ELSE
79
80                MESSAGE "No customer has been chosen. ",
81                   "Use the Query option to select ",
82                   "a customer."
83
84                NEXT OPTION "Query"
85
```

```
 86           END IF
 87
 88
 89       COMMAND "Delete" "Delete a customer." HELP 4
 90
 91           IF chosen THEN
 92
 93               PROMPT "Are you sure you want to ",
 94                  "delete this customer (y/n)? "
 95                  FOR CHAR answer
 96
 97                  IF answer = "y" THEN
 98
 99                     CALL delete_row()
100
101                  END IF
102
103           ELSE
104
105               MESSAGE "No customer has been chosen. ",
106                  "Use the Query option to select ",
107                  "a customer."
108
109               NEXT OPTION "Query"
110
111           END IF
112
113
114       COMMAND "Exit" "Leave the CUSTOMER menu." HELP 5
115
116           EXIT MENU
117
118   END MENU
119
120END FUNCTION
121
122
123FUNCTION enter_row()
124
125   LET int_flag = 0
126
127   MESSAGE ""
128
129   CLEAR FORM
130
131   INPUT p_customer.fname THRU p_customer.phone
132       FROM sc_cust.*
133
134   IF int_flag THEN
135       LET int_flag = FALSE
136       ERROR "Customer entry aborted."
137       RETURN
138   END IF
139
140   LET p_customer.customer_num =  0
141
142   INSERT INTO customer VALUES (p_customer.*)
143
144   LET p_customer.customer_num = SQLCA.SQLERRD[2]
145
146   DISPLAY p_customer.customer_num TO customer_num
```

```
147
148    MESSAGE "Row added."
149
150    SLEEP 3
151
152    MESSAGE ""
153
154END FUNCTION
155
156
157
158FUNCTION query_data()
159
160    DEFINE
161        where_clause CHAR(200),
162        sql_stmt CHAR(250),
163        answer CHAR(1),
164        exist SMALLINT
165
166    LET int_flag = 0
167
168    MESSAGE ""
169
170    CLEAR FORM
171
172
173    MESSAGE "Enter search criteria and press ESC."
174
175    SLEEP 3
176
177    MESSAGE ""
178
179    CONSTRUCT where_clause on customer.* FROM customer_num,
180        fname, lname, company, address1, address2, city, state,
181        zipcode, phone
182
183        IF int_flag THEN
184            LET int_flag = FALSE
185            ERROR "Customer query aborted"
186            RETURN
187        END IF
188

189        LET sql_stmt = "SELECT * FROM customer where ",
190            where_clause clipped
191
192    PREPARE ex_sel FROM sql_stmt
193
194    DECLARE q_curs CURSOR FOR ex_sel
195
196    LET exist = FALSE
197
198    LET chosen = FALSE
199
200    FOREACH q_curs INTO p_customer.*
201
202        LET exist = TRUE
203
204        DISPLAY BY NAME p_customer.*
205
206        PROMPT "Enter 'y' to select this customer ",
```

```
207           "or RETURN to view next customer:  "
208           FOR CHAR answer
209
210           IF answer = "y" THEN
211
212               LET chosen = TRUE
213
214               EXIT FOREACH
215
216           END IF
217
218    END FOREACH
219
220    IF exist = FALSE THEN
221
222        MESSAGE "No customer rows found."
223
224        SLEEP 3
225
226        MESSAGE ""
227
228    ELSE
229
230        IF chosen = FALSE THEN
231
232            MESSAGE "There are no more customer rows."
233
234            SLEEP 3
235
236            MESSAGE ""
237
238            CLEAR FORM
239

240        END IF
241
242    END IF
243
244END FUNCTION
245
246
247FUNCTION change_data()
248
249    LET int_flag = 0
250
251    INPUT p_customer.fname THRU p_customer.phone
252        WITHOUT DEFAULTS FROM sc_cust.*
253
254    IF int_flag THEN
255        LET int_flag = FALSE
256        ERROR "Customer update aborted."
257        RETURN
258    END IF
259
260    UPDATE customer
261        SET customer.* = p_customer.*
262        WHERE customer_num = p_customer.customer_num
263
264    MESSAGE "Row updated."
265
266    SLEEP 3
```

```
267
268    MESSAGE ""
269
270END FUNCTION
271
272
273FUNCTION delete_row()
274
275    LET int_flag = 0
276
277    DELETE FROM customer WHERE customer_num =
278        p_customer.customer_num
279
280    CLEAR FORM
281
282    MESSAGE "Row deleted."
283
284    SLEEP 3
285
286    MESSAGE ""
287
288END FUNCTION
```

The following notes pertain to this example:

- The GLOBALS section defines the global record **p_customer** with variables corresponding to the columns of the **customer** table. It also defines a flag called **chosen** that indicates whether the user has selected a customer.
- The MAIN section issues the DEFER INTERRUPT command, opens and displays the **customer** form, and calls the **show_menu** function.
- The **show_menu** function displays the program menu with **Add**, **Query**, **Modify**, and **Delete** options and calls the functions to carry out the actions described by these options.
- The **enter_row** function assigns values to the variables of the **p_customer** record from the data entered by the user onto the screen form. It assigns a value of zero to the **p_customer.customer_num** variable and inserts the new row into the **customer** table.
- The **query_data** function uses the CONSTRUCT statement to perform a query by example, declares the cursor **q_curs**, and retrieves rows into the **p_customer** program record by means of a FOREACH loop. The user can select a row for updating or deleting.
- The **change_data** function allows the user to update a customer row returned by the **query_data** function and previously selected.
- The **delete_row** function allows the user to delete a customer row returned by the **query_data** function and previously selected.

Compiling the Form and Program

In order to debug an 4GL program, you must first compile the program and any forms that it displays.

To compile the screen form from the Programmer's Environment

1. Enter `r4gl` at the system prompt to access the **INFORMIX-4GL** menu.
2. Choose the **Form** option from the **INFORMIX-4GL** menu.
3. Choose the **Compile** option from the **FORM** menu.
4. Select the **customer** form.

   You see the message:

   ```
   Form compilation in progress...please wait while the
   form compiles.
   ```
5. Choose the **Exit** option to return to the **INFORMIX-4GL** menu.

To compile the program from the Programmer's Environment

1. Enter `r4gl` at the system prompt to access the **INFORMIX-4GL** menu.
2. Choose the **Module** option from the **INFORMIX-4GL** menu.
3. Choose the **Compile** option from the **MODULE** menu.
4. Select the **customer** program.
5. Choose the **Runable** option from the **COMPILE MODULE** menu.

   You see the message

   ```
   Compilation in progress... please wait while the
   program compiles.
   ```

Operating the Debugger

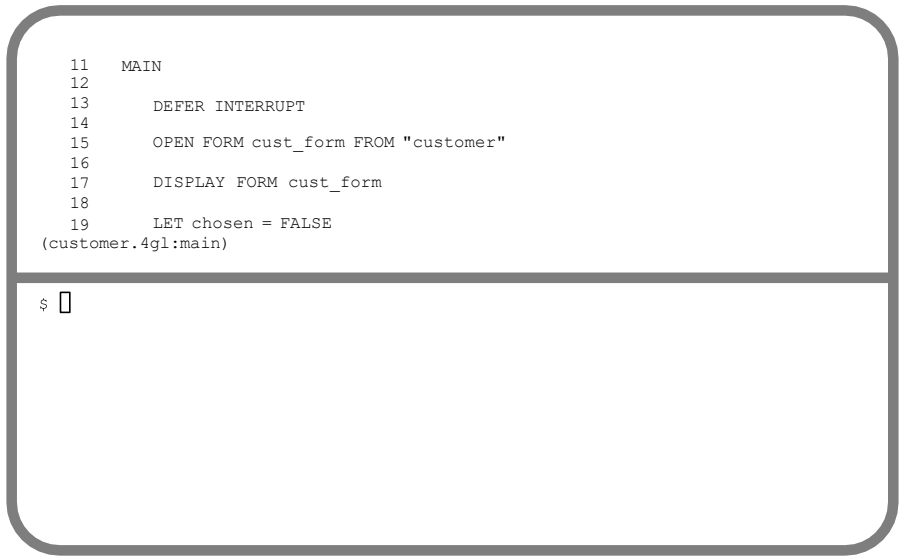
Once the program is compiled, you are ready to access the Debugger.

To initiate the debugging session from the Programmer's Environment

1. Choose the **Debug** option from the **MODULE** menu.
2. Choose the **customer** program.

   After a few seconds, the Source and Command windows appear on the terminal screen, as shown in Figure 2-1

Figure 2-1
The Debugger Screen



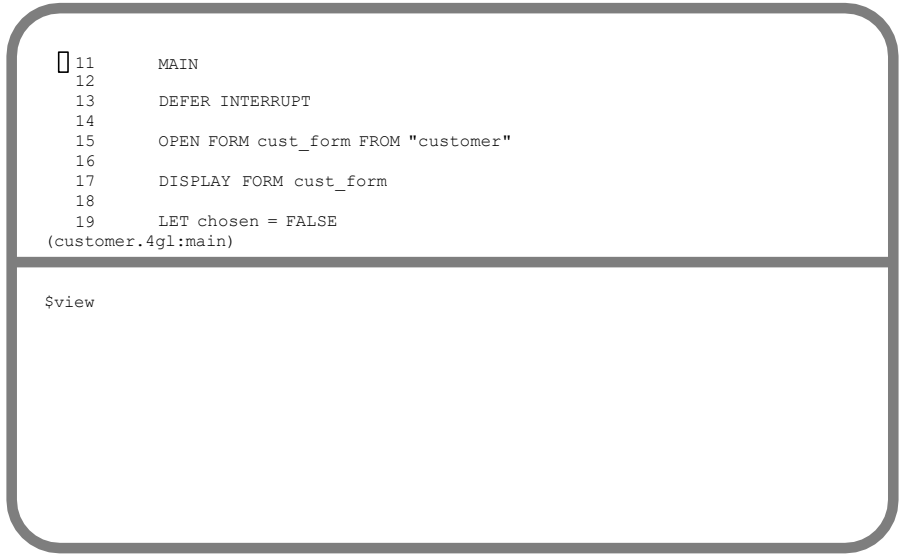
Viewing the Source Window

On a standard terminal screen, the first nine lines of the MAIN section appear in the Source window, followed by the name of the module and current function. The Debugger always displays the MAIN block when you initiate a session, even if it is preceded in the module by a DATABASE or GLOBALS statement. The cursor is in the Command window, which is ready to accept input.

The VIEW Command

You use the VIEW command to move the cursor to the Source window and examine the source module. VIEW is the only command that allows you to move the cursor from the Command window to the Source window.

Figure 2-2
The VIEW Command



You can enter any Debugger command using the smallest number of characters that uniquely identify it. For example, you can enter `view` as `vi`.

The examples and screens in this manual identify Debugger commands by their full names for ease of reference. You can, however, use the abbreviated forms to save keystrokes. "Short Forms of Keywords" on page 9-24 lists all of the shortest unique forms of the Debugger commands. Help is available by typing `help` or `help` *`command`*.

Returning to the Command Window

You can use the **Interrupt** key (usually DEL or CTRL-C) to return the cursor to the Command window.

In addition to using the **Interrupt** key, you can use any alphabetic key or sequence of alphabetic keys to return to the Command window, which saves you keystrokes because you can enter Debugger commands while the cursor is still in the Source window. For example, typing RUN and pressing RETURN while the cursor is in the Source window both returns the cursor to the Command window and executes the RUN command.

This feature is controlled by the EXITSOURCE terminal display parameter. If you change the value of EXITSOURCE from ON to OFF, the **Interrupt** key is the only key that returns the cursor to the Command window. See Chapter 8, "The Debugging Environment," for more information on using EXITSOURCE.

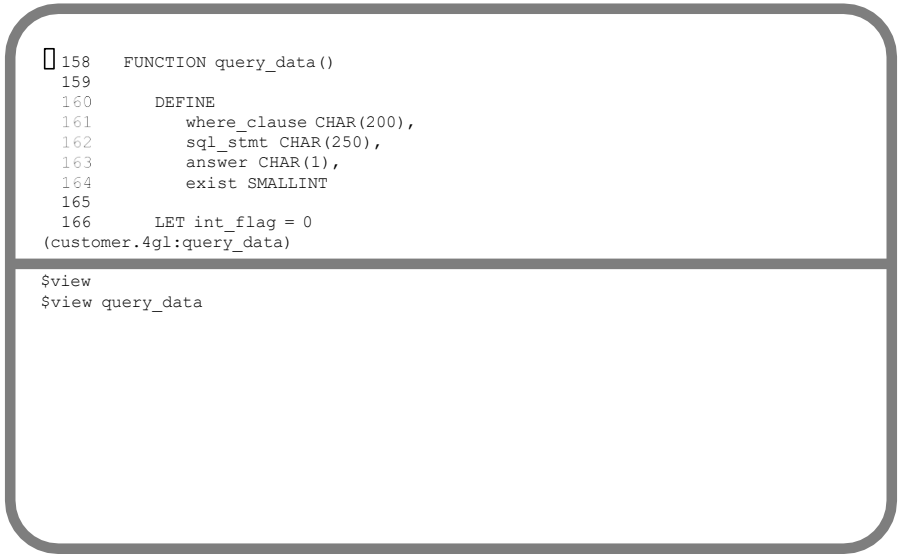
Viewing a Specific Function

You can view a specific function by following the VIEW command with the function name. For example:

```
VIEW query_data
```

The Source window scrolls to display the function you have indicated, as shown in Figure 2-3 on page 2-14. Do not follow the function name with parentheses or an argument list.

Figure 2-3
Viewing a Specific Function



Scrolling the Source Window

When you enter the VIEW command, the cursor moves to the first line of the Source window. To scroll through this window you can use the UP ARROW and DOWN ARROW keys, or the following CTRL keys.

Control Key	Description
CTRL-K	Moves the cursor up one line
CTRL-J	Moves the cursor down one line
CTRL-B	Moves the cursor up one window
CTRL-F	Moves the cursor down one window
CTRL-U	Moves the cursor up one-half window
CTRL-D	Moves the cursor down one-half window

As you scroll, the last line of the Source window changes when necessary to reflect the function currently displayed.

You can move to a specific line of the source module by typing the line number and pressing RETURN. Typing `$` in the Source window moves the cursor to the last line of the source module.

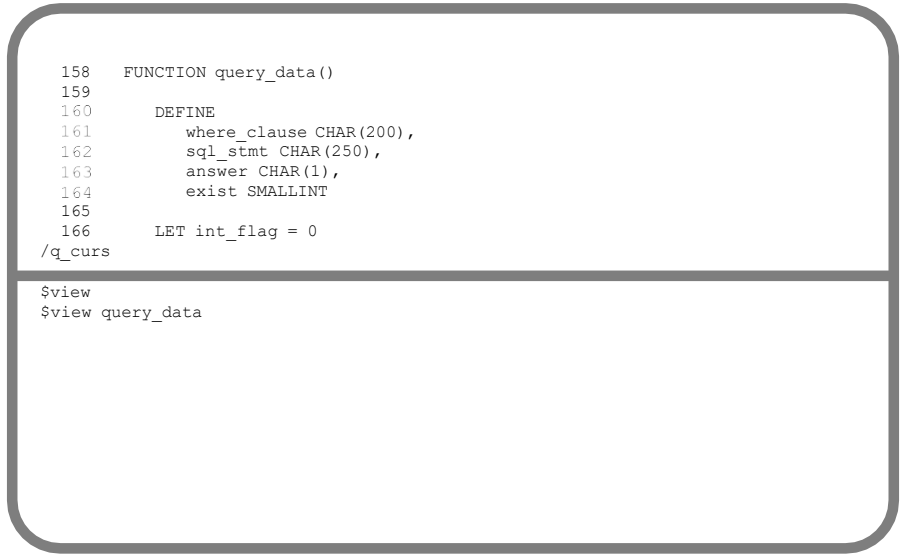
Searching for Patterns of Characters

The following keys are used to search the Source window for a particular pattern of characters.

Search Character	Description
/	Searches forward from the current position
?	Searches backward from the current position

For example, you can enter `/q_curs` to search forward in the source module for the first occurrence of the string `q_curs`.

Figure 2-4
Searching for a Pattern in the Source Window



Pressing RETURN moves the cursor to the next occurrence of the string in the direction indicated.

Using Wildcards

Within a search string, you can use the following wildcards to search for a partial match.

Wildcard Character	Description
*	Matches any subset of a string
?	Matches any single character
[*x*-*y*]	Matches any character between *x* and *y* in the ASCII collating sequence

For example, the following entry causes the Debugger to search forward in the Source module for any pattern bounded by the capital letters *C* and *D*:

```
/C*D
```

This search pattern finds RECORD, COMMAND, and SQLCA.SQLERRD[2].

The next entry causes the Debugger to search backward in the source module for a pattern beginning with a lowercase letter in the range *c-f* and ending with the characters *_row*:

```
?[c-f]*_row
```

This search pattern finds **delete_row()** and **enter_row()**.

To search for a string using wildcards

1. Use the keys described in this section to scroll and search through the Source window.
2. Press the **Interrupt** key when you are finished to return to the Command window.

Working with the Command Window

In the Command window, you enter commands at the **$** prompt, and then press RETURN. On a standard terminal, the Command window can display 10 lines at once, and the information in the window scrolls upward automatically when the window is filled. Output from your commands and error messages are also displayed in this window. The 50 most recent command lines remain in a buffer, and you can examine them with many of the same tools you use to manipulate the Source window. In this section, you use the LIST command to generate output to the Command window, and learn the similarities and differences in working with the two windows.

The LIST Command

When you first set up a debugging session with the **customer** program, the Command window is empty. You can use the LIST command to display the following features of the debugging environment:

- The terminal display state
- Tracepoints
- Breakpoints

If you do not specify any options, the LIST command displays all current debugging parameters.

The following example shows the output of the LIST command:

```
autotoggle            on
displaystops          on
sourcetrace           off
exitsource            on
printdelay            off
timedelay source      1
timedelay  command    0
source  lines         9
command  lines        10
```

Listed first are the default values for the five terminal display parameters AUTOTOGGLE, DISPLAYSTOPS, SOURCETRACE, EXITSOURCE, and PRINTDELAY. The activity of the TIMEDELAY command can affect either the Source or Command window, and the LIST command displays the values for both. Listed next are the default sizes of the Source and Command windows. There are presently no tracepoints or breakpoints.

In this chapter, you learn how to use the SOURCETRACE parameter to monitor program statements as they execute, and how to use the TIMEDELAY command with the SOURCE option to control the speed at which SOURCETRACE executes. Chapter 6 illustrates the use of the AUTOTOGGLE parameter. A complete description of all the parameters and commands is provided in Chapter 8.

Scrolling the Command Window

You can scroll the lines in the command buffer using either the UP ARROW or DOWN ARROW key, or the same six CTRL keys used for scrolling the Source window. See "Scrolling the Source Window" on page 2-14 for a summary of these CTRL keys.

You can move forward a specific number of lines by typing the number and pressing RETURN. (In contrast, typing a number and pressing RETURN in the Source window moves the cursor to the line indicated.)

Searching in the Command Window

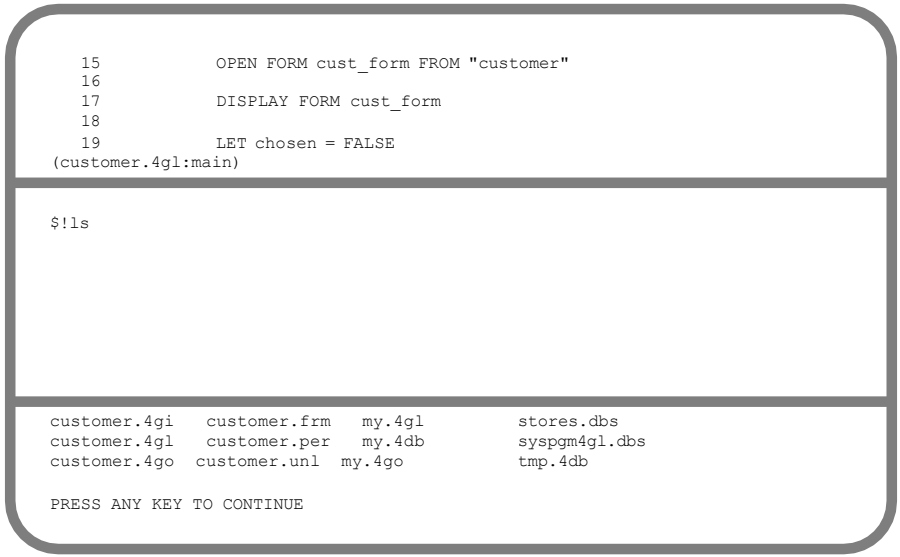
As in the Source window, you can use the forward slash (/) and the question mark (?) to search for a pattern of characters within the Command window. When you enter the forward slash in the Command window, the Debugger searches forward in the command buffer rather than from the current $ prompt. Pressing RETURN takes you to the next line of the buffer rather than to the next occurrence of the search pattern.

You can use the same wildcards to locate a partial match in the Command window as you can in the Source window. See "Using Wildcards" on page 2-16 for a summary of these wildcards.

Executing Operating System Commands

You can execute operating system commands from either the Source window or the Command window by preceding them with the exclamation point (!). For example, enter `!ls` at the $ prompt to display the files in the current directory.

Figure 2-5
The Command Window



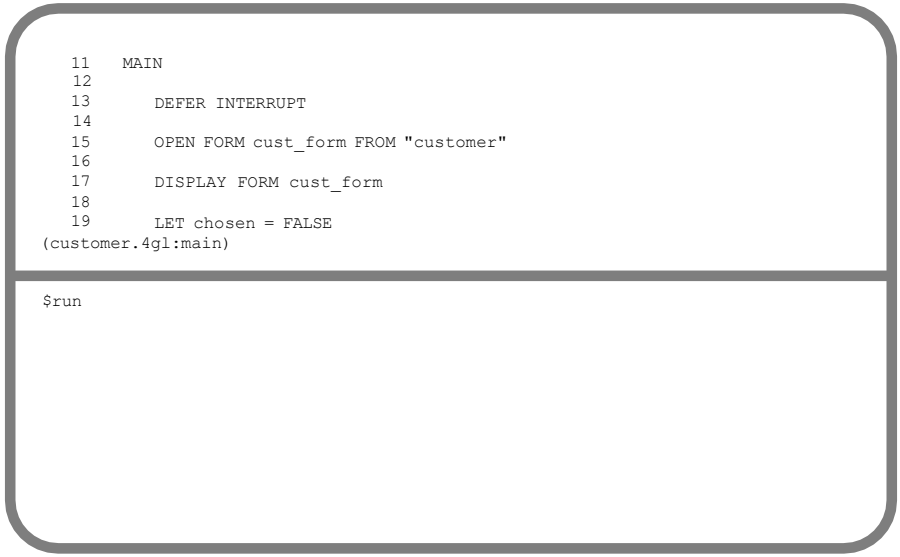
To work with the Command window

1. Use the keys described in this section to scroll and search through the Command window.
2. Press any key when prompted to return to the Command window.

Starting the Debugger

You start operation of the Debugger by entering the RUN command or by pressing F5. Running a 4GL program with all terminal display parameters in their default states, and with no tracepoints or breakpoints set, is very similar to running the program outside the Debugger.

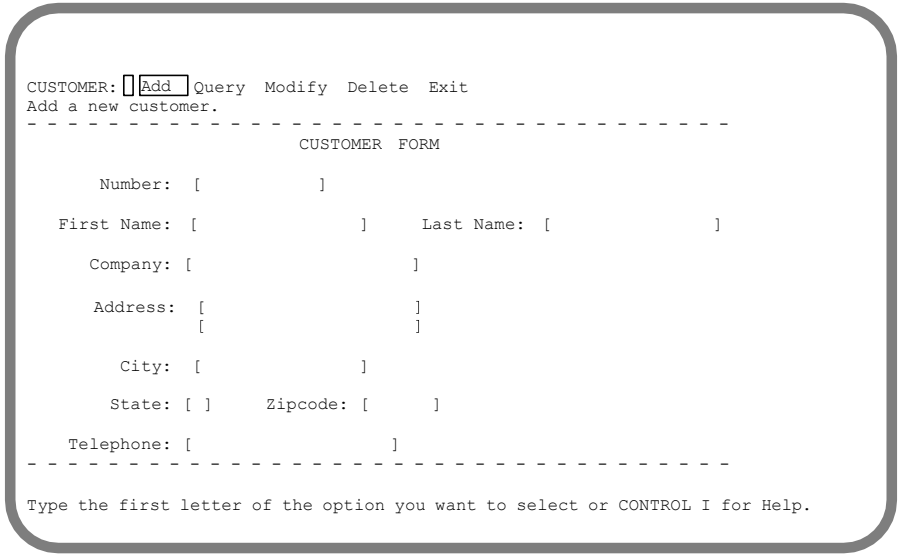
Figure 2-6
Starting the Debugger



The Application Screen

As soon as you enter the RUN command, program execution begins. The Debugger switches immediately to the Application screen as the program displays the form and the menu, as shown in Figure 2-7.

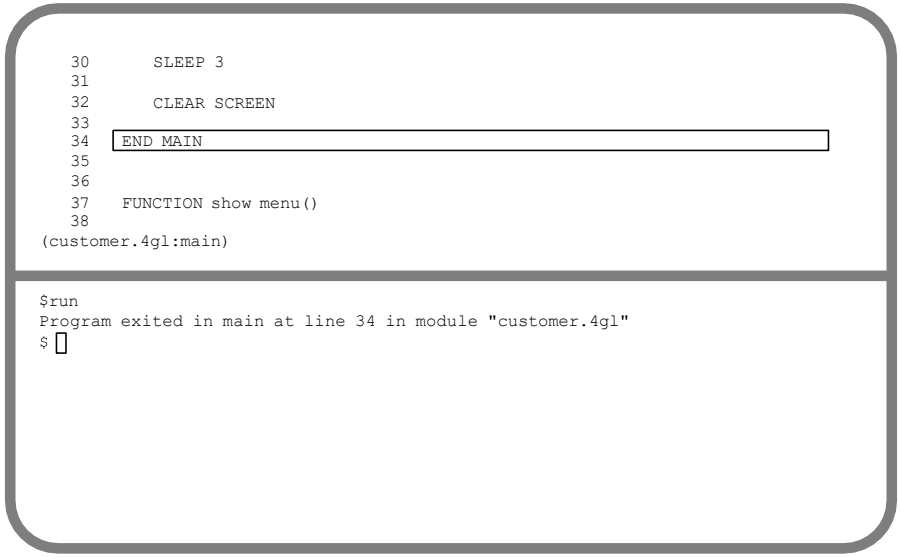
Figure 2-7
The Application Screen with the customer Program



You can choose any of the options presented and work with the program exactly as if it were executing outside the Debugger.

Selecting the **Exit** option terminates execution of the program but does not stop operation of the Debugger. The Source and Command windows reappear on the terminal screen, as shown in Figure 2-8.

Figure 2-8
Terminating the Program



The Source window highlights the statement at which execution terminates and displays the block of code in which this statement appears. In the current example, the final statements of the MAIN section execute following the return from the **show_menu** function.

The Command window displays the name of the function and the line number at which program execution terminated. The cursor appears on the following line. You can rerun the program at this point or enter any other valid Debugger command.

The Terminal Display State

The Debugger provides five terminal display parameters that allow you to control the interaction of the Debugger windows and of the Debugger and Application screens. Refer to "The LIST Command" on page 2-17 for a listing of the parameters with their default values. You use the TURN ON and TURN OFF commands to alter the values of these parameters. Following is a description of the SOURCETRACE parameter. The TIMEDELAY command with the SOURCE option is also described because it affects the execution of SOURCETRACE. Chapter 8 provides a complete description of all the parameters and commands that determine the terminal display state.

SOURCETRACE

The default value for SOURCETRACE is OFF. If SOURCETRACE is turned on, the Debugger highlights each line of code as it executes, and modifies the contents of the Source window accordingly. There is consequently more interaction between the Debugger and Application screens. The Source and Command windows remain on the terminal screen when you begin program execution, and the Debugger switches to the Application screen only when the program requires input from the user.

Turning on SOURCETRACE increases the time required for a debugging session. It can be very useful, however, when you are first becoming familiar with a program or when you are working with a program that is relatively small. Alternatively, it can be turned on for those sections of a program that require particular attention and turned off otherwise.

TIMEDELAY SOURCE

The TIMEDELAY command with the SOURCE option controls the speed with which SOURCETRACE executes. TIMEDELAY is similar to the 4GL SLEEP command. The default value for TIMEDELAY SOURCE is `1`. If you would like to highlight each line of source code for an additional second, you can enter the TIMEDELAY command as follows:

```
timedelay source 2
```

Running with SOURCETRACE

To observe the operation of the Debugger with the SOURCETRACE parameter set to ON, enter the command:

```
turn on sourcetrace
```

and then enter `run` to restart the **customer** program.

When operation begins, the Debugger windows remain on the terminal screen. A highlighting bar moves through the Source window line by line, beginning with the first executable statement in the MAIN block. This statement is line 13:

```
13   DEFER INTERRUPT
```

The Source window scrolls upward as necessary to display the currently executing statement.

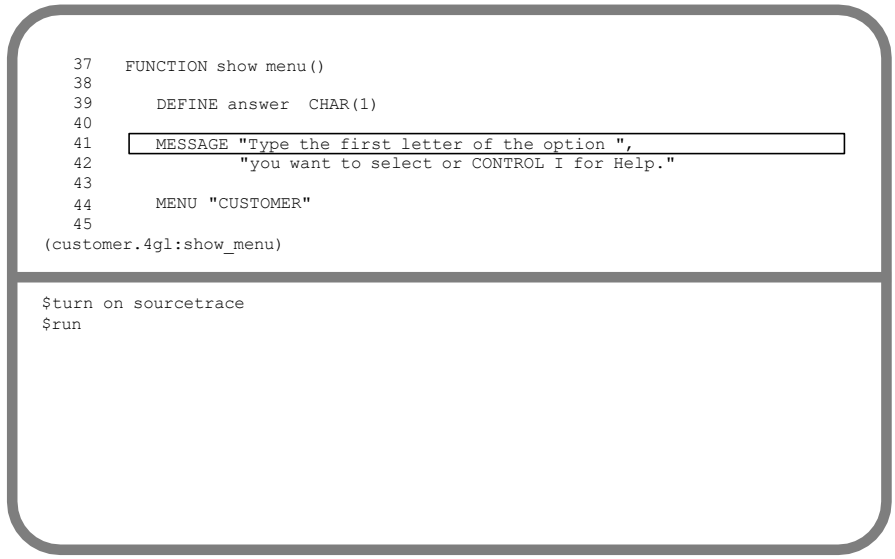
Important: *The following statements in a 4GL program are not executable and are not highlighted by SOURCETRACE: DEFINE, MAIN, FUNCTION declarations, GLOBALS, LABEL, and any comments you have included in your program.*

The first function call occurs at line 26 with the following statement:

```
26   CALL show_menu()
```

Figure 2-9 illustrates the appearance of the Source window immediately after this statement executes.

Figure 2-9
Source Window After Executing show_menu()

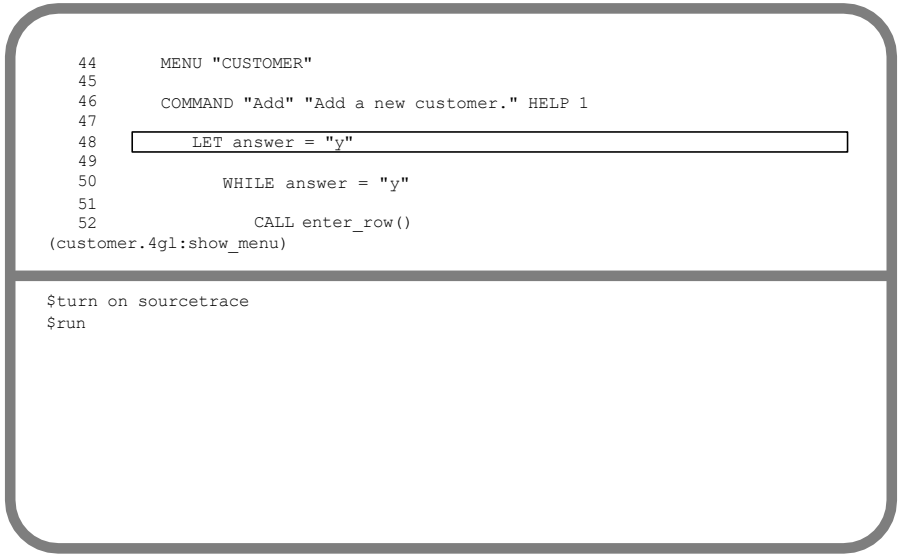


The first nine lines of the **show_menu** function now appear in the Source window, and the last line of the Source window has changed to reflect the fact that **show_menu** is the current function.

As soon as the MENU statement executes, the program requires input from the user. The Debugger switches to the Application screen and waits for you to choose one of the menu options. Your choice determines the next program segment to be executed.

Choose the **Add** option to enter a new customer. The Debugger redisplays the Debugger screen with the **show_menu** function still current and traces the execution of the statements specified for this option.

Figure 2-10
Debugger Screen

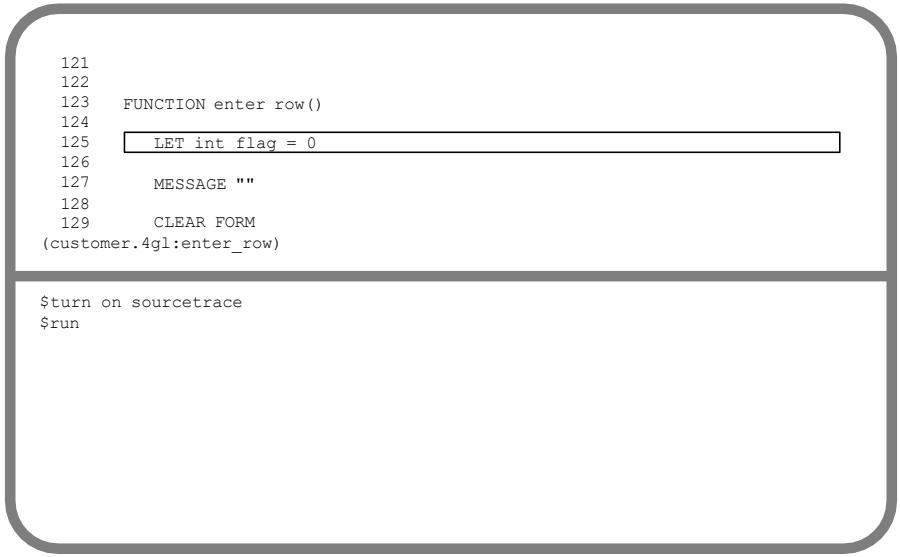


When the Debugger executes line 52:

```
52   CALL enter_row()
```

the first nine lines of the new function appear in the Source window, and the last line of the Source window changes to reflect the fact that **enter_row** is the current function.

Figure 2-11
Source Window with enter_row as the Current Function



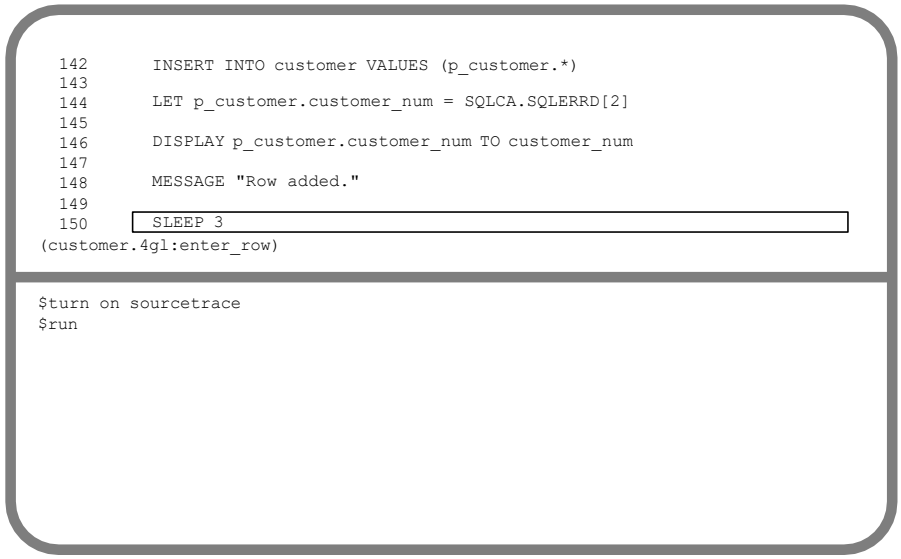
At line 131, the **customer** program again requires input:

```
131   INPUT  p_customer.fname  THRU  p_customer.phone
132      FROM  sc_cust.*
```

The Debugger redisplays the Application screen and waits for you to enter the new customer information. At this point, you can enter values for a new customer in the form and press ESC or RETURN after the last field to terminate your entry.

As soon as input terminates, the Debugger redisplays the Source and Command windows. The highlight bar moves line by line through the remaining statements of the **enter_row** function.

Figure 2-12
Highlight Bar Moving Through enter_row Function



At line 154, program control returns to the calling function **show_menu**:

```
154   END FUNCTION
```

The Debugger executes the statement beginning at line 54:

```
54   PROMPT "Do you want to "
55      "enter another row (y/n) ? "
56      FOR CHAR answer
```

and switches to the Application screen for your response. Your response to this prompt determines the next program segment to be executed. If you choose `y`, the program reexecutes the WHILE loop and recalls the **enter_row** function. If you choose `n`, the program exits from the WHILE loop and redisplays the program menu.

To complete this example

1. Enter `n` at the prompt to return to the program menu.
2. Choose one or more of the remaining options and continue observing the interaction of the Debugger and Application screens.
3. Choose the **Exit** option when you are finished to end the program and return the cursor to the Command window.

Restoring the Environment

The debugging parameters that you set while running the Debugger for a particular session are saved and restored automatically as long as you do not leave the Programmer's Environment and as long as you do not access the Debugger with a different program. The following features are automatically restored:

- The current terminal display state
- Search path of 4GL source files
- Tracepoints
- Breakpoints
- Programmer-defined aliases

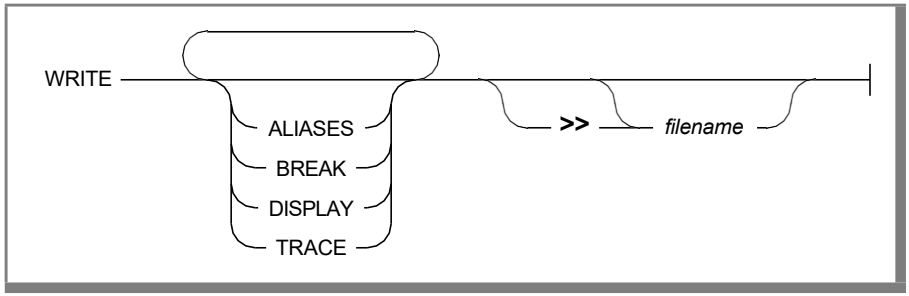
You can, for example, exit from the Debugger, work with other menu options such as FORM or PROGRAM, and reaccess the Debugger with the **customer** program without losing the established debugging environment.

Saving the Environment

You can save your debugging parameters in a file for use in a subsequent session. You can save all or some combination of the parameters listed in the previous section.

The WRITE Command

The following diagram shows the syntax of the WRITE command.



You can use the WRITE command to save debugging parameters in a file. If you do not specify an option, the WRITE command saves all of the current debugging parameters. If you do not specify a filename, the WRITE command creates or appends to a file with the same name as the current program. The Debuggeraddstheextension **.4db** toanyfilecreatedwiththe WRITEcommand.

Enter `write` to save the current debugging environment.

Figure 2-13
Saving the Current Debugging Environment

```
   26       CALL show_menu()
   27
   28       MESSAGE "End program."
   29
   30       SLEEP 3
   31
   32       CLEAR SCREEN
   33
   34   END MAIN
(customer.4gl:main)
```

```
$turn on sourcetrace
$run
Program exited in main at line 34 in module "customer.4gl"
$write
$
```

In this example, WRITE saves the terminal display parameters and the TIMEDELAY command options with their current values, and places them in the **customer.4db** file.

The customer.4db File

When you use the WRITE command to save the debugging environment for this session, the Debugger creates a file named **customer.4db**. The values in this file are restored automatically whenever you access the Debugger and initiate a new debugging session with the **customer** program.

To view the customer.4db file

1. Enter an exclamation point (!), followed by the appropriate system command (such as **cat**, **page**, or **more**) to view the contents of **customer.4db**.

   For example:

   ```
   !cat customer.4db
   ```

2. Press any key when done to return the cursor to the Command window.

   The following list shows the contents of this file:

   ```
   alias f1 = help
   alias f2 = step
   alias f3 = step into
   alias f4 = continue
   alias f5 = run
   alias f6 = list break trace
   alias f7 = list
   alias  f8  =  dump
   alias  f9  =  exit
   use = .
   turn on autotoggle
   turn on sourcetrace
   turn on displaystops
   turn on exitsource
   turn off printdelay
   timedelay source 1
   timedelay command 0
   list display
   ```

The function keys are defined with the ALIAS command, in a file named **init.4db.** This file is placed in the **$INFORMIXDIR/etc** directory when you install the Debugger software. These function keys are part of the debugging environment and are included in the output of the WRITE command. Chapter 8 provides more information on the system **init.4db** file.

The USE command allows you to tell the Debugger what directories to search to locate your 4GL source files. This information is part of the debugging environment and is included in the output of the WRITE command. Because you have not entered the USE command in the current session, the Debugger searches only the current directory. The current directory is symbolized by a period following the equal sign (=.).

All of the terminal display parameters are saved with their default values with the exception of SOURCETRACE, which you have turned on. The two options of the TIMEDELAY command are also saved with default values.

The current sizes of the Source and Command windows are not saved by WRITE unless you have changed the default sizes with the GROW command. These values are included in the output of the LIST command as SOURCE LINES and COMMAND LINES, respectively. Refer to "The LIST Command" on page 2-17 to review the default values. See Chapter 9 for information on the GROW command.

The final line in the file is the command:

```
list display
```

This command is appended by the Debugger to every **.4db** file created with the WRITE command that includes terminal display characteristics. The LIST DISPLAY command indicates that the Debugger should automatically display these characteristics to the Command window when the environment is restored.

If you do not use the WRITE command to save all or some part of the current operating environment, no **.4db** file is produced, and you will need to reestablish the environment in future sessions.

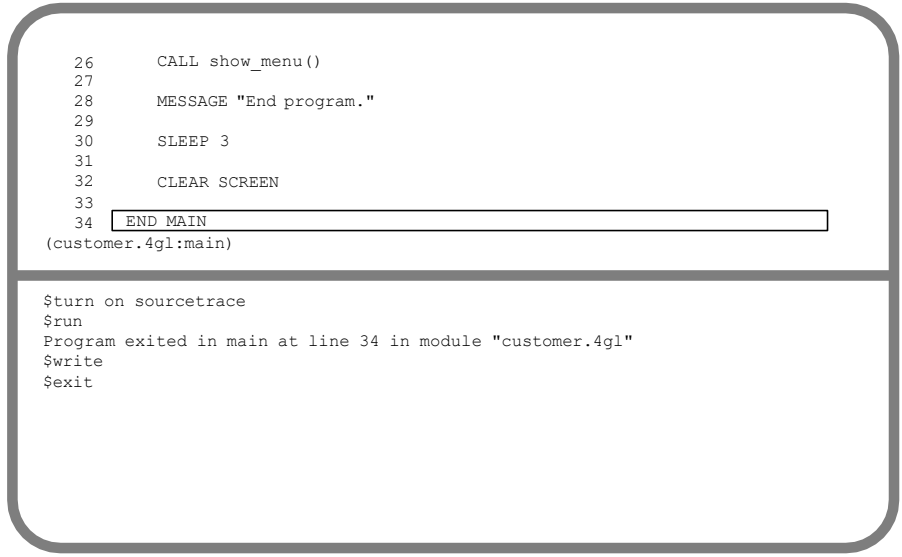
Alternatively, you can specify a different name for the file. In this case, you specifically need to tell the Debugger to read the contents of the file during the session.

Chapter 8 provides more information on the **.4db** files and their use.

Exiting from the Session

You use the EXIT command to stop operation of the Debugger. If you are running the Debugger from the Programmer's Environment, entering `exit` returns you to the **MODULE** or **PROGRAM** menu.

Figure 2-14
The EXIT Command



You must successfully compile a 4GL program before you can debug it. When a program is running under the Debugger, the terminal display toggles between the Debugger and the Application screens. While the Debugger screen is displayed, you can monitor the execution of your source code in the Source window and enter commands in the Command window. The Command window also displays output from your commands. You can scroll the contents of both windows using CTRL keys and the UP ARROW and DOWN ARROW keys.

The Application screen allows you to work with your program as if it were running outside the Debugger. Terminal display parameters control the interaction of Debugger windows and the Debugger and Application screens. The debugging parameters that you set during a session are restored automatically while you remain within the Programmer's Environment. In addition, you can save these parameters in a file for use at some other time.

Chapter

3

Tracing Logic of the customer Program

In This Chapter

You are now acquainted with the debugging environment and know how to monitor INFORMIX-4GL statements as they execute. This chapter introduces you to an important and powerful debugging tool, the setting of tracepoints. The following topics are covered:

- How the debugging environment is restored when you start a new session
- How to use the TRACE command to set tracepoints
- How to use the DUMP command to see the values of global and local variables in the current function
- How to interrupt a program and resume execution from the point of interruption
- How to combine commands
- How to use the NOTRACE command to remove a tracepoint

The examples in this chapter use the same sample program, form, and help file as those in the previous chapter.

Restoring the Environment

The Debugger provides many tools to monitor the execution of a 4GL program. You can, for example, trace functions as they execute, or trace the values of variables as they change. You can list the functions that have been called to arrive at the current statement. You can interrupt your program, change the value of a variable or variables, and continue execution with the new values. Any of these methods provides valuable insights into the operation of a program, and there is no single best approach in any particular case. The examples in this chapter show you how to use the TRACE and DUMP commands to learn more about **customer.4gl**.

To view restored values from the **customer** program, access the Debugger and select the **customer** program as in the previous chapter. When the Debugger screen appears, the following information scrolls through the Command window:

```
Current search path: .

TERMINAL DISPLAY STATE
autotoggle            on
displaystops          on
sourcetrace           on
exitsource            on
printdelay            off
timedelay source      1
timedelay command     0
source lines          9
command lines         10
```

All these values, with the exception of SOURCE LINES and COMMAND LINES, are restored from the **customer.4db** file that you created with the WRITE command at the end of the previous chapter. All the parameters and commands appear with their default values except SOURCETRACE, which you saved with a value of `on`.

Tip: *If you have not left the Programmer's Environment since carrying out the debugging steps in Chapter 2, "Getting Started with the Debugger," these values are restored not from* ***customer.4db*** *but from a temporary file that remains in existence as long as you are running 4GL. The same information is displayed whether the environment is restored from a temporary file or from a* ***.4db*** *file. If you left the Programmer's Environment and did not issue the WRITE command at the end of Chapter 2, no values are restored or displayed.*

The TRACE Command

The TRACE command allows you to set a tracepoint when any of the following situations occurs:

- A particular line of code executes
- The value of a specific variable changes
- A specific function executes
- Any function in the program executes

When the Debugger encounters a tracepoint, it displays information in the Command window or, optionally, in a file. This information includes the line number, function name, and module name where the tracepoint was reached, as well as any optional instructions you have specified. Program execution continues automatically.

The following sections illustrate the different options for setting tracepoints, with examples from **customer.4gl**.

Tracing a Line Number

The simple format for setting a tracepoint at a line number is as follows:

```
TRACE lineno
```

For example, to set a tracepoint at line number 192:

```
192   PREPARE ex_stmt FROM sel_stmt
```

you would enter the following command:

```
trace 192
```

This command causes the Debugger to record when line 192 executes, and to display the module and function name in which this line occurs.

If the statement you are tracing spans several lines, the tracepoint is set at the first line in the statement. If you enter a line number that does not designate an executable statement, the tracepoint is set instead at the first executable statement following the line number. For example, if you set a tracepoint at line 160:

```
160   DEFINE
161     where_clause  CHAR(200),
162     sql_stmt  CHAR(250),
163     answer  CHAR(1),
164     exist  SMALLINT
165
166   LET  int_flag  =  0
167
168
```

the tracepoint actually occurs at line 166 because DEFINE is not an executable statement:

```
166 LET  int_flag  =  0
```

If your program consists of more than one module, and the module name is not specified, the Debugger sets the tracepoint in the module currently displayed in the Source window. See Chapter 5, "A Multi-Module Program: cust_order," for information on setting tracepoints in a multi-module program.

Tracing a Variable

The simple format for tracing a variable is as follows:

```
TRACE  variable
```

To trace when the value of the global variable **chosen** changes, enter the following command:

```
trace  chosen
```

This command causes the Debugger to output the new value every time the value of **chosen** changes between `0` (FALSE) and `1` (TRUE). It also records the line number, module name, and function name at which each change occurs.

You must qualify a global variable with the keyword GLOBAL if there is a local variable with the same name in the function currently displayed in the Command window. To set a tracepoint on a local variable, you must either make its function the current function by displaying it in the Source window, or qualify the variable with the function name.

For example, if you want to set a tracepoint on the variable **exist** in the **query_data** function, you must either use the VIEW command to display the **query_data** function in the Source window or specify the function in which the variable appears using one of the following conventions:

```
trace  (query_data)  exist
```

or

```
trace   function.query_data.exist
```

Chapter 5 contains more information on specifying the scope of reference of variables.

Tracing a Function

The simple format for tracing a function is as follows:

```
TRACE  function
```

To trace when the **change_data** function executes, you can enter the following command:

```
trace  change_data
```

This command causes the Debugger to output the line numbers where the function is called and where it returns. The Debugger also records any parameters passed to or returned by the function. Chapter 6, "Tracing Logic of the cust_order Program," provides an example of a parameter returned by a traced function.

Do not use parentheses when setting a tracepoint on a function.

Tracing All Functions

You can use the FUNCTIONS keyword to trace all functions in the current program. Because functions are often nested, tracing all functions can provide a convenient map of the flow of program control.

To trace all functions in a program, enter the following command:

```
trace functions
```

When you trace all functions, the Debugger records when each function begins execution and when each terminates. It also records any parameters passed to or returned by these functions.

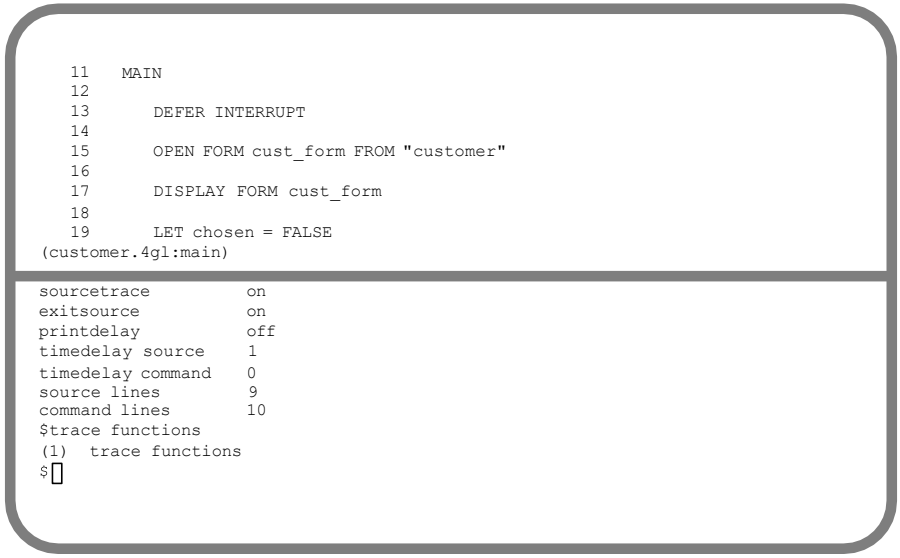
This chapter illustrates tracing all the functions in the **customer** program.

Setting a Tracepoint

When you define a tracepoint, the Debugger assigns it a reference number and displays this number in the Command window. For example, you can enter `trace functions` to trace all functions in the **customer** program.

Figure 3-1 illustrates your entry of this command and the Debugger response.

Figure 3-1
Setting a Tracepoint



The Debugger assigns this tracepoint a reference number of (1).

Outputting to a File

When the Debugger encounters a tracepoint, it generates output to the Command window. Alternatively, you can designate a file to receive the output by using the symbol >> and specifying a filename. If a file with this name does not currently exist, the Debugger creates it. If a file with this name already exists, the Debugger appends the output from the tracepoint to it. To redirect the output of the command TRACE FUNCTIONS to the file **session1**, you would enter the following command:

```
trace functions >> session1
```

While deciding whether or not to send output from Debugger commands to a file, keep in mind that the Command window holds the 50 most recent lines in a buffer, which can be scrolled. If you anticipate that your commands will generate more than 50 lines of output, or if you want a record of the session after it has ended, specify a filename.

Running with a Tracepoint

You are now ready to begin program execution with the tracepoint you have set.

To execute a program with a tracepoint

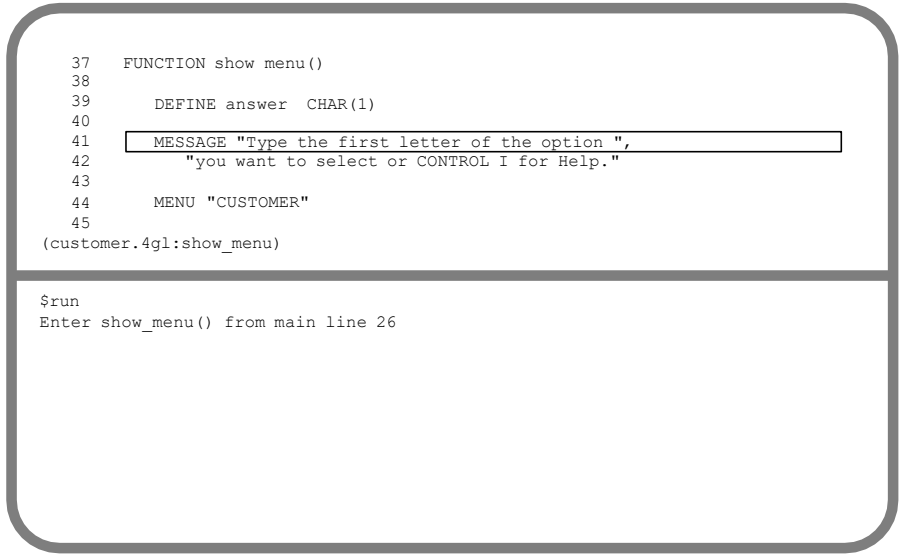
1. Enter the RUN command or press F5 to start operation of the Debugger.

   Because the value of SOURCETRACE is ON, the Source window highlights each statement as it executes. The tracepoint is first reached at line 26 in the MAIN section:

   ```
   26   CALL show_menu()
   ```

   The Debugger enters the **show_menu** function and records the tracepoint, as shown in Figure 3-2.

Figure 3-2
Entering the show_menu Function



   When the Debugger executes the MENU statement at line 44, it switches to the Application screen and waits for you to make a selection.

2. Choose the **Query** option.

   When you choose the **query_data** function. The Debugger modifies the contents of the Command window to register the new occurrence of the tracepoint, as shown in Figure 3-3.

Figure 3-3
Entering the query_data Function

```
  169
  170       CLEAR FORM
  171
  172
  173       MESSAGE "Enter search criteria and press ESC."
  174
  175       SLEEP 3
  176
  177       MESSAGE ""
(customer.4gl:query_data)
```

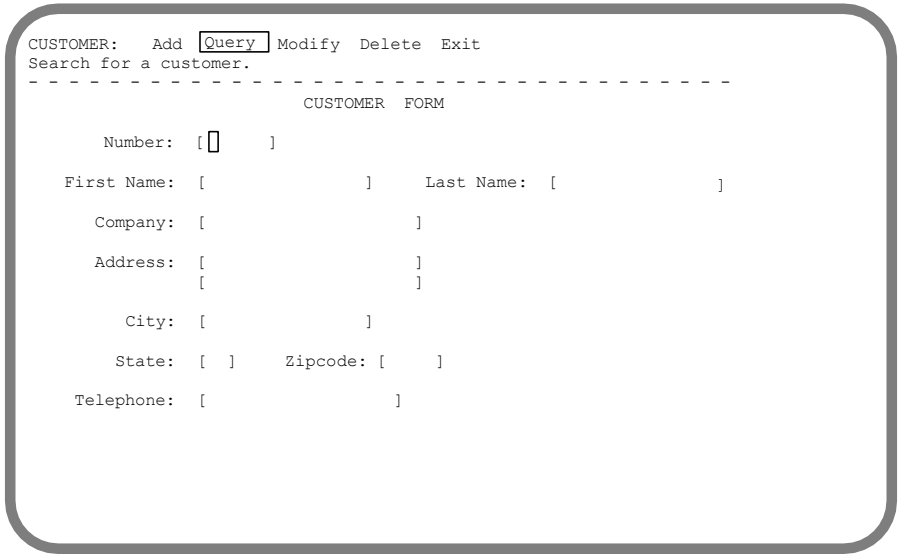
```
timedelay command    0
source lines         9
command lines        10
$trace functions
(1)  trace functions
$run
Enter show_menu() from main line 26
Enter query data() from show menu line 64
```

When the Debugger executes the statement at line 179:

```
179  CONSTRUCT where_clause ON customer.* FROM customer_num,
180    fname, lname, company, address1, address2, city, state,
181    zipcode, phone
```

it redisplays the Application screen and waits for you to enter search criteria, as shown in Figure 3-4.

Figure 3-4
Entering Search Criteria



3. Enter the search criterion for customers whose last name matches the pattern `B*` and press ESC to terminate your query.

   If a query returns no rows, the **query_data** function terminates automatically, and program control returns to the calling function, **show_menu**. If a query returns one or more rows, the function terminates when you enter `y` in response to the following prompt:

   ```
   Enter 'y' to select this customer or RETURN
   to view next customer:
   ```

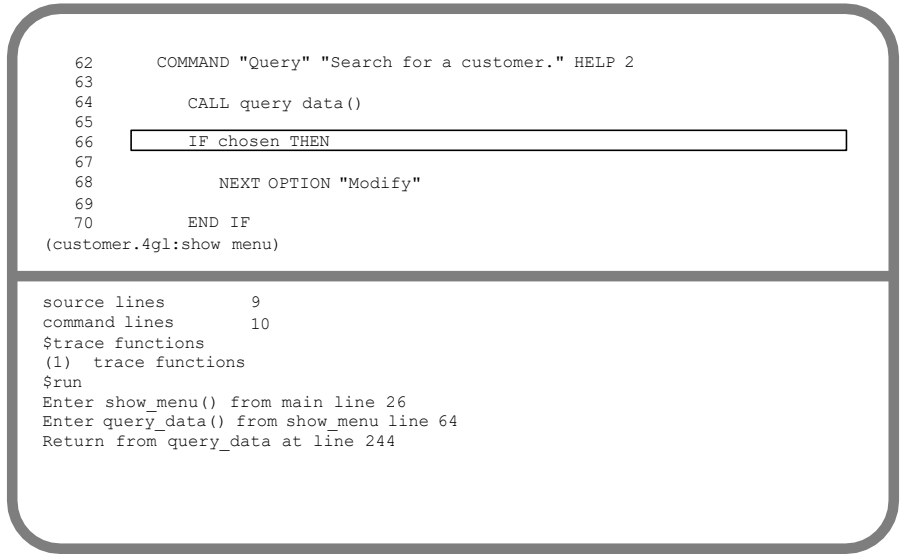
   or when the active set is exhausted.

4. Press RETURN, if necessary, in response to this prompt until the record for customer Dick Baxter is displayed on the form.

5. Enter `y` to select this customer.

   When you make your entry, the remaining statements in the **query_data** function execute, and the Debugger modifies the contents of the Command window to record the return from the function, as shown in Figure 3-5.

Figure 3-5
Returning from the query_data Function

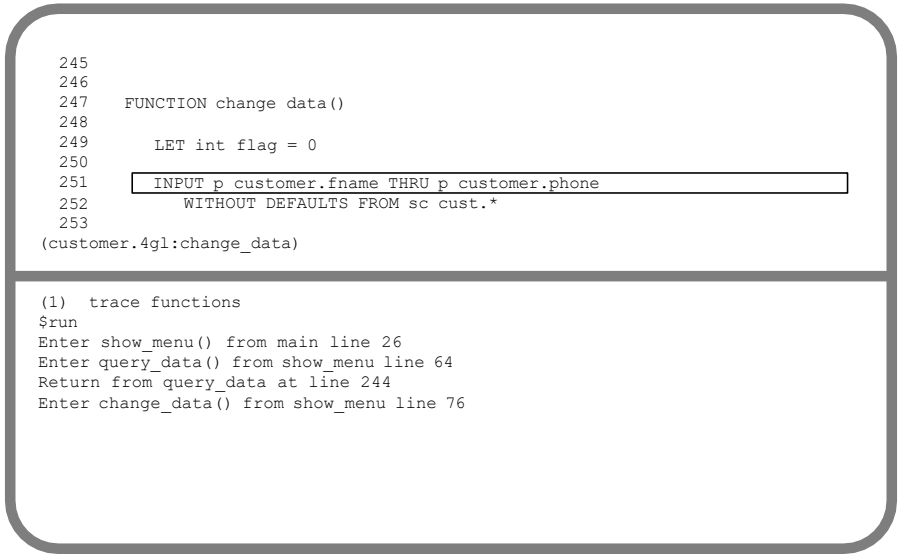


The row you have selected can be updated or deleted.

6. Choose the **Modify** option to update the values for the customer currently displayed on the form.

   The program calls the **change_data** function, and the Debugger provides new output to the Command window.

Figure 3-6
Entering the change_data Function



When the Debugger executes the statement at line 251:

```
251   INPUT p_customer.fname THRU p_customer.phone
         WITHOUT DEFAULTS FROM sc_cust.*
```

it redisplays the Application screen and waits for you to change one or more data values.

7. Update the current customer record and press ESC to commit your changes.

   The **change_data** function terminates after updating the values for the customer in the database, and program control returns to the calling function. The Debugger records the return from the **change_data** function, as shown in Figure 3-7.

Figure 3-7
Returning from the change_data Function

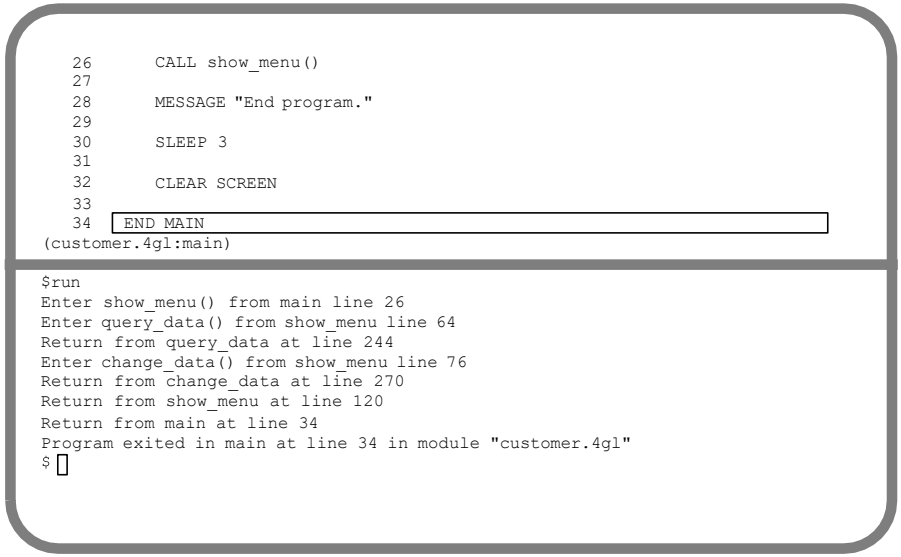
```
   40
   41       MESSAGE "Type the first letter of the option ",
   42           "you want to select or CONTROL I for Help."
   43
   44       MENU "CUSTOMER"
   45
   46       COMMAND "Add" "Add a new customer." HELP 1
   47
   48          LET answer = "y"
(customer.4gl:show_menu)
```

```
$trace functions
(1)  trace functions
$run
Enter show_menu() from main line 26
Enter query_data() from show_menu line 64
Return from query_data at line 244
Enter change_data() from show_menu line 76
Return from change_data at line 270
```

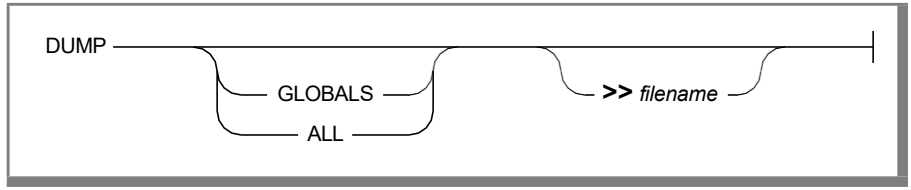
8. Choose the **Exit** option to terminate the program.

   The appearance of the Debugger screen when you end execution of the program is as shown in Figure 3-8.

Figure 3-8
Exiting the Program



The DUMP Command



The DUMP command provides an easy way to monitor the values of all global, module, and local variables in the currently executing function. The DUMP command has two options. If you do not specify GLOBALS or ALL, the Debugger displays the values of all local variables in the currently executing function.

The GLOBALS Option

If you specify GLOBALS, the Debugger displays the values of all programmer-defined global and module variables in the currently executing function, as well as the values of the following 4GL global variables:

- **int_flag**
- **quit_flag**
- **status**
- the elements of the **SQLCA** record

The ALL Option

If you specify ALL, the Debugger displays the values of all global, module, and local variables in the currently executing function.

Dumping to a File

You can redirect the output of the DUMP command to a file with the >> symbol. Write the output of the DUMP command to a file if you anticipate that this output will exceed 50 lines, or if you want a record after the session has ended. The following command redirects the output of the DUMP command to the file named **session1**:

```
dump globals >> session1
```

Executing the DUMP Command

In order for you to use the DUMP command, your program must be running. You must, therefore, do one of the following things:

- Interrupt the program, enter the DUMP command, and continue operation.
- Combine DUMP with another command such as TRACE.

The following section shows how to interrupt program execution and display the current values of program variables. The section "Combining Commands" on page 3-25 shows how to combine the TRACE and DUMP commands to list the values of variables automatically as tracepoints are reached.

Interrupting Program Execution

An important feature of the debugging process is the ability to interrupt a program and resume execution from the point of interruption. You can interrupt the execution of a program at any time by pressing the **Interrupt** key (usually DEL or CTRL-C) on your terminal. The cursor returns to the Command window, which can accept any valid Debugger command.

Interrupting a Program Versus Interrupting the Debugger

When a 4GL program is running outside the Debugger, an interrupt entered by the user terminates execution unless you have issued a DEFER INTERRUPT command. If you have issued the DEFER INTERRUPT command, 4GL sets the value of the global variable **int_flag** to `1`, or TRUE, and takes the action you have specified. See the *INFORMIX-4GL Reference* for more information on the DEFER INTERRUPT command.

When a 4GL program is running under the Debugger, however, pressing the **Interrupt** key suspends program execution and returns control of the Debugger to you. The Source window highlights the next statement to execute when operation resumes. The cursor returns to the Command window, which can accept any valid instruction.

Entering an Interrupt

The following procedure describes how to interrupt execution of the **customer** program and dump the current values of all program variables.

To enter an interrupt

1. Rerun the **customer** program.
2. Choose the **Query** option.
3. Enter the search criterion `Redwood City` in the **City** field and press ESC.
4. Press RETURN, if necessary, in response to the prompt until the information for Anthony Higgins is retrieved and displayed, as shown in Figure 3-9.

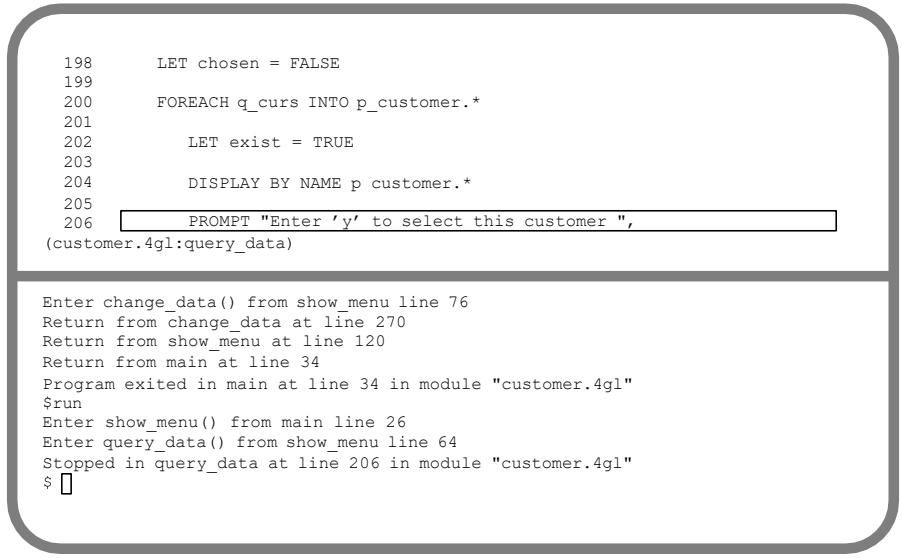
Figure 3-9
Customer Information for Anthony Higgins

```
CUSTOMER:   Add  Query  Modify  Delete  Exit
Search for a customer.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                          CUSTOMER  FORM

       Number:  [104         ]

   First Name:  [Anthony         ]     Last Name:  [Higgins         ]

      Company:  [Play Ball!           ]

      Address:  [East Shopping Cntr. ]
                [422 Bay Road        ]

         City:  [Redwood City    ]

        State:  [CA]    Zipcode:  [94026]

    Telephone:  [415-368-1100       ]
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Enter 'y' to select this customer or RETURN to view next customer:
```

5. Press the **Interrupt** key (usually DEL or CTRL-C) on your terminal to suspend execution of the program and return the cursor to the Command window.

   The Debugger registers the interrupt, as shown in Figure 3-10.

Figure 3-10
Debugger Showing an Interrupt



6. Enter the following command to view the current values of the program variables:

```
dump all
```

The Debugger lists both global and local variables in the **query_data** function in response to this command.

Examining Global Variables

The Debugger displays the values of all global variables in the current function under the heading DUMPING GLOBAL VARIABLES. The following code example illustrates the values of the global variables at this point in the execution of the program. A brief description of each variable follows the example.

```
DUMPING  GLOBAL  VARIABLES

p_customer       =       {
     customer_num  =  104
     fname  =  "Anthony          "
     lname   =   "Higgins                "
     company  =  "Play Ball!             "
     address1  =  "East  Shopping  Cntr.  "
     address2  =  "422  Bay  Road          "
     city  =  "Redwood City      "
     state  =  "CA"
     zipcode  =  "94026"
     phone   =   "415-368-1100        "
}
chosen  =  0
status  =  0
int_flag  =  0
quit_flag  =  0
sqlca  =  {
     sqlcode  =  0
     sqlerrm  =  (null)
     sqlerrp  =  (null)
     sqlerrd  =  {0,  0,  5,  0,  0,  18}
     sqlawarn  =  "        "
}
```

p_customer

The values currently assigned to the members of the **p_customer** record are those of Anthony Higgins.

chosen

The value of **chosen** is 0, or FALSE, indicating that no customer has been selected at this point.

status

The value of **status** is 0, indicating that the most recent statement that sets this variable has successfully executed. 4GL updates the value of **status** when an SQL or form-related statement executes.

int_flag

The value of **int_flag** is 0, or FALSE. The value of **int_flag** is FALSE, even though you have pressed the **Interrupt** key. This is because an interrupt entered during operation of the Debugger returns control of the Debugger to you. No interrupt signal is sent to the program. To send an interrupt to the program and set the value of **int_flag** to 1, you must use the CONTINUE INTERRUPT command. This command is described in the section "Sending an Interrupt to a Program" on page 3-29.

quit_flag

The value of **quit_flag** is 0, or FALSE, indicating that no quit signal has been sent to the program. See the *INFORMIX-4GL Reference* for more information on the **quit_flag** variable and the DEFER QUIT command.

The SQLCA Record

The last global variables to be listed are the elements of the **SQLCA** record.

The value of **sqlcode** is set to 0, indicating that the most recent SQL statement has successfully executed. 4GL uses this member of the **SQLCA** record to update the value of **status** every time an SQL statement executes.

The variables **sqlerrm** and **sqlerrp** are not implemented at this time and appear in the output as NULL.

Two of the six elements of the array **sqlerrd** have values following execution of the query, and those that have not been assigned values appear as 0. The third element, **sqlerrd[3]**, is set to 5, corresponding to the five rows retrieved by the query. The last element, **sqlerrd[6]**, is set to 18. This is the row ID of the last row processed. Because the field on which you have queried, **City**, is not indexed, this value is the row ID of the last customer in the table.

Tip: *If you have added or deleted customer rows while working with the program, the values of* ***sqlerrd[3]*** *and* ***sqlerrd[6]*** *might be different. These elements of the* ***sqlerrd*** *array are principally used following database inserts.*

The value of **sqlawarn** is blank because no warnings were received in the execution of the query.

See the *INFORMIX-4GL Reference* for more information on the **SQLCA** record and its members.

Examining Local Variables

Following the display of global variables, the Debugger lists the values of the local variables in the current function under the heading DUMPING LOCAL VARIABLES OF FUNCTION **[query_data]**. There are four local variables in the function:

```
DUMPING LOCAL VARIABLES OF FUNCTION [query_data]

where_clause = "customer.city="Redwood City"

                                        "

sql_stmt = "SELECT * FROM customer where customer.city="Redwood City"


               "
answer = (null)
exist = 1
```

where_clause

The first variable, **where_clause**, holds the clause constructed by 4GL when the user presses ESC to indicate that all search criteria have been entered. The **where_clause** variable is a CHARACTER variable of length 200.

sql_stmt

The variable **sql_stmt** holds the string that is formed by concatenating the SELECT statement with **where_clause**. The **sql_stmt** variable is a CHARACTER variable of length 250.

answer

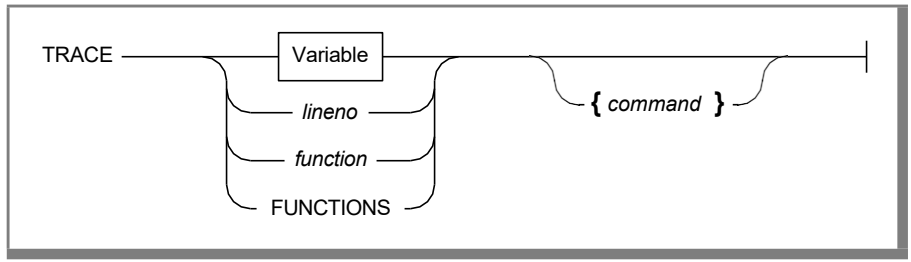
The value of **answer** is NULL, indicating that you have not yet pressed `y` or RETURN in response to the prompt on this iteration of the FOREACH loop.

exist

The value of **exist** is set to `1`, or TRUE, because the FOREACH loop has successfully retrieved at least one row.

Combining Commands

You can specify the action to be taken when a tracepoint is reached by placing Debugger commands in curly braces following the definition of the tracepoint. The general format for specifying commands is as follows.



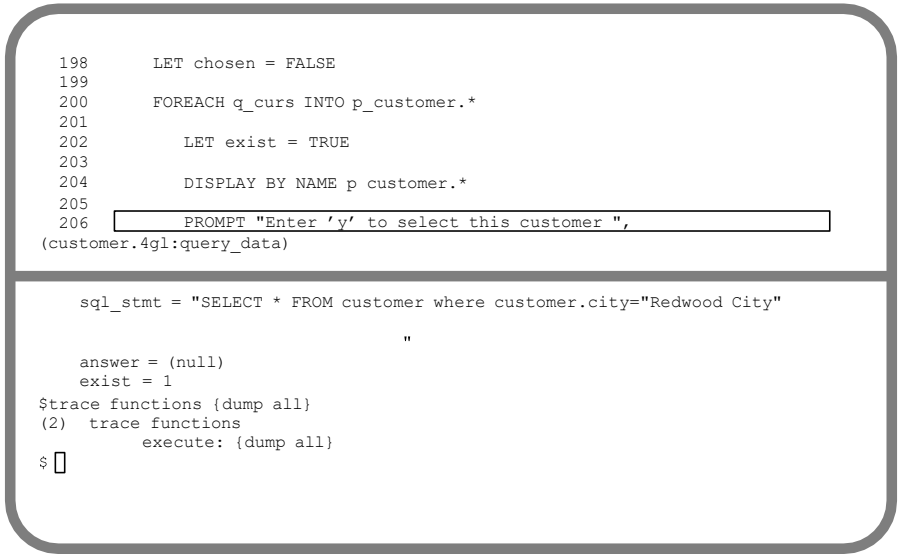
In the current example, you can combine the TRACE and DUMP commands to trace the functions of **customer.4gl**, printing out the current values of all global and local variables automatically as each function executes.

Set a new tracepoint as follows:

```
trace functions { dump all }
```

Figure 3-11 illustrates your entry of this command and the response from the Debugger.

Figure 3-11
Combining the TRACE and DUMP Commands



Because there is one existing tracepoint:

```
(1)  trace  functions
```

the Debugger assigns the new tracepoint a reference number of (2). The keyword **execute** indicates the action the Debugger will take when it reaches the tracepoint.

Removing Tracepoints

It is not possible to edit an existing tracepoint. If you want to modify the command, you must remove the tracepoint and set another one.

You use the NOTRACE command to remove a tracepoint. You can refer to tracepoints by their reference numbers when removing them. The format for removing a tracepoint by its reference number is as follows:

```
NOTRACE  refno
```

See Chapter 9 for more information on the NOTRACE command and its options.

To remove a tracepoint

1. Enter the following command to remove the first tracepoint you have set, (1) **trace functions**:

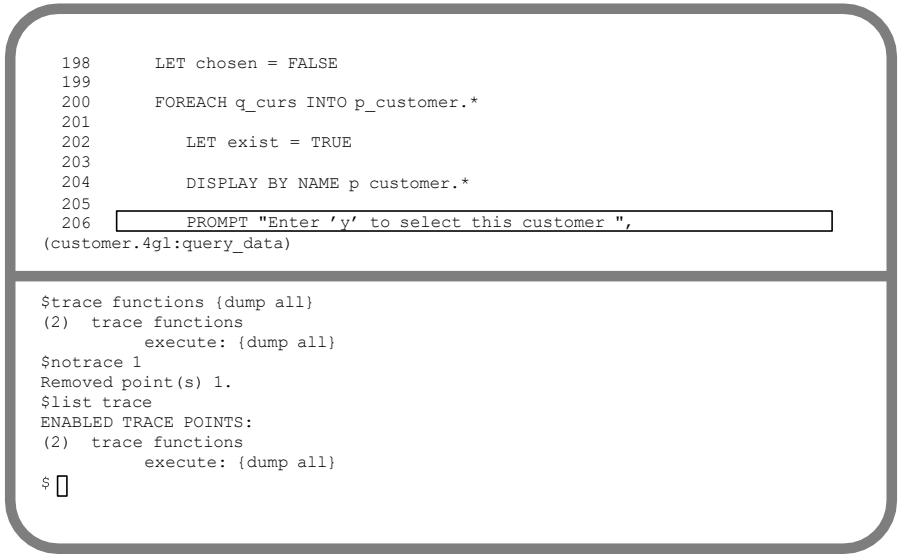
   ```
   notrace  1
   ```

2. Confirm that tracepoint (2) is now the only existing tracepoint by entering the LIST command as follows:

   ```
   list  trace
   ```

   The appearance of the Command window when you have made your entries is as shown in Figure 3-12.

Figure 3-12
Command Window



Because tracepoint (2) is currently active, it appears here as ENABLED. Chapter 4, "Analyzing a Logical Error in the customer Program," shows you how to use the DISABLE command to deactivate a tracepoint without removing it. You are now ready to resume operation of the program with tracepoint (2).

The CONTINUE Command

You use the CONTINUE command to resume operation of a program. CONTINUE restarts the program from the point of interruption rather than from the beginning.

In the present example, you interrupted execution while the PROMPT statement at line 206 was awaiting input. This is the point at which execution resumes.

Sending an Interrupt to a Program

You can use the CONTINUE command with the INTERRUPT option to bypass the Debugger and send an interrupt directly to your program. For example, you would want to do this in order to test those sections of your code that handle interrupts entered by the user.

If you have included the DEFER INTERRUPT statement in your program, issuing the command:

```
continue  interrupt
```

sets the value of the global variable **int_flag** to `1`, or TRUE.

If you have not included the DEFER INTERRUPT statement in your program, issuing the CONTINUE INTERRUPT command aborts program execution and returns control to the Debugger.

Chapter 9 provides more information on CONTINUE INTERRUPT.

Entering the CONTINUE Command

You are ready to resume execution with the new tracepoint. Now the Debugger will output the values of all global and local variables in a function as soon as the function is entered.

To enter the CONTINUE command

1. Enter the CONTINUE command, or press F4, to resume execution of the program.

   When you make your entry, the Debugger redisplays the Application screen with the values for Anthony Higgins still current. The cursor is at the prompt, and the program is awaiting input. The appearance of the Application screen is as shown in Figure 3-13.

Figure 3-13
Continuing Execution Following an Interrupt

```
CUSTOMER:   Add  Query  Modify  Delete  Exit
Search for a customer.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                          CUSTOMER  FORM

       Number:  [104         ]

   First Name:  [Anthony         ]     Last Name:  [Higgins         ]

      Company:  [Play Ball!           ]

      Address:  [East Shopping Cntr. ]
                [422 Bay Road        ]

         City:  [Redwood City    ]

        State:  [CA]    Zipcode:  [94026]

    Telephone:  [415-368-1100       ]
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Enter 'y' to select this customer or RETURN to view next customer:
```

2. Enter y in response to the prompt to select this customer.

   The Debugger toggles briefly to the Debugger screen as it registers the return from the **query_data** function. It then redisplays the Application screen with the **Modify** option highlighted.

3. Choose the **Modify** option to update the current record.

   When you make your selection, the program calls the **change_data** function. The Debugger registers the start of the new function in the Command window and automatically outputs the current values of all global variables in this function. There are no local variables in the function.

   The following display illustrates the output to the Command window at this occurrence of the tracepoint:

```
Enter change_data from show_menu line 76
DUMPING GLOBAL VARIABLES

    p_customer      =      {
        customer_num = 104
        fname = "Anthony          "
        lname  =  "Higgins                "
        company = "Play Ball!             "
        address1 = "East Shopping Cntr.  "
        address2 = "422 Bay Road          "
        city = "Redwood City     "
        state = "CA"
        zipcode = "94026"
        phone  =  "415-368-1100       "
    }
    chosen = 1
    status = 0
    int_flag = 0
    quit_flag = 0
    sqlca = {
    sqlcode = 0
    sqlerrm = (null)
    sqlerrp = (null)
    sqlerrd = {0, 0, 0, 0, 0, 18}
    sqlawarn = "       "
    }
```

   You observe that the values currently assigned to the **p_customer** program record are those of Anthony Higgins. The value of **chosen** is set to `1`, indicating that this customer has been selected.

4. Modify one or more values for this customer and press ESC to commit your changes.

5. Choose the **Exit** option to end the program and return control to the Debugger.

Saving and Exiting

Tracepoints and breakpoints that you have defined during a debugging session are saved and restored automatically if you remain in the 4GL Programmer's Environment and if you do not initiate a debugging session with a different program. You can use the WRITE command with the TRACE option to save tracepoints in a file for use in future debugging sessions with the **customer** program.

There is currently one active tracepoint, **trace functions { dump all }**.

To save and exit from the Debugger

1. Enter the following command to save this tracepoint as part of the debugging environment:

   ```
   write trace
   ```

   The Debugger appends the tracepoint to the file **customer.4db**. The contents of this file now include the following debugging parameters:

   ```
   alias f1 = help
   alias f2 = step
   alias f3 = step into
   alias f4 = continue
   alias f5 = run
   alias f6 = list break trace
   alias f7 = list
   alias  f8  =  dump
   alias  f9  =  exit
   use = .
   turn on autotoggle
   turn on sourcetrace
   turn on displaystops
   turn on exitsource
   turn off printdelay
   timedelay source 1
   timedelay command 0
   list display
   trace  functions  {dump  all}
   ```

2. Enter the EXIT command or press F9 to end the debugging session and return to the Programmer's Environment.

Tracepoints are an excellent way to familiarize yourself with the flow of control of unfamiliar programs. You can set tracepoints to monitor when particular statements or functions execute or when the values of variables change. When a tracepoint is reached, the Debugger displays relevant information regarding the tracepoint in the Command window. This information includes the function name, line number, and module name at which the tracepoint occurred. If you are tracing a function, the Debugger displays any values passed to or returned by the function. Program execution resumes automatically.

The DUMP command allows you to display the values of local, global, and module variables in the currently executing function. You can combine the TRACE and DUMP commands so that the listing of variables occurs automatically when tracepoints execute.

You can interrupt a program at any time by pressing the **Interrupt** key. Pressing the **Interrupt** key returns control of the Debugger to you so that you can perform such activities as viewing the current values of program variables. You can use the CONTINUE command to resume execution of the program from the point of interruption.

Chapter

4

Analyzing a Logical Error in the customer Program

In This Chapter

By now you are familiar with the general operation of the Debugger and with some of the tools available to you for tracing program logic. You have used these tools to gain insight into the operation of the **customer.4gl** program.

A bug has been coded into this program that causes it to produce undesirable results in two circumstances. This chapter illustrates the use of the Debugger to discover the source of the problem, and introduces a new and important diagnostic tool, the setting of breakpoints. The following topics are covered:

- How to use the BREAK command to suspend program execution when a specific point in the program is reached or a specific condition is met
- How to use the DISABLE command to deactivate tracepoints or breakpoints during a debugging session
- How to use the PRINT command to display the value of a variable
- How to use the LET command to change the value of a variable
- How to use the STEP command to execute one or more INFORMIX-4GL statements
- How to use the NOBREAK command to permanently remove a breakpoint

The examples in this chapter are based on the same sample program, form, and help file as in the previous two chapters.

Observing Problems with the Program

An intentional bug causes problems with the **customer** program if either of the following situations occurs:

- The **Delete** menu option is chosen two or more times in succession.
- The **Delete** and **Modify** menu options are chosen in sequence.

This section shows you how to produce these problems while running the program outside the Debugger. Following this, you will access the Debugger and use it to discover the cause of the problems.

To run the program from the outside the debugger

1. Enter `r4gl` at the system prompt.
2. Choose the **Module** option from the **INFORMIX-4GL** menu.
3. Choose the **Run** option from the **MODULE** menu.
4. Select the **customer** program.

Choosing Delete Twice in Succession

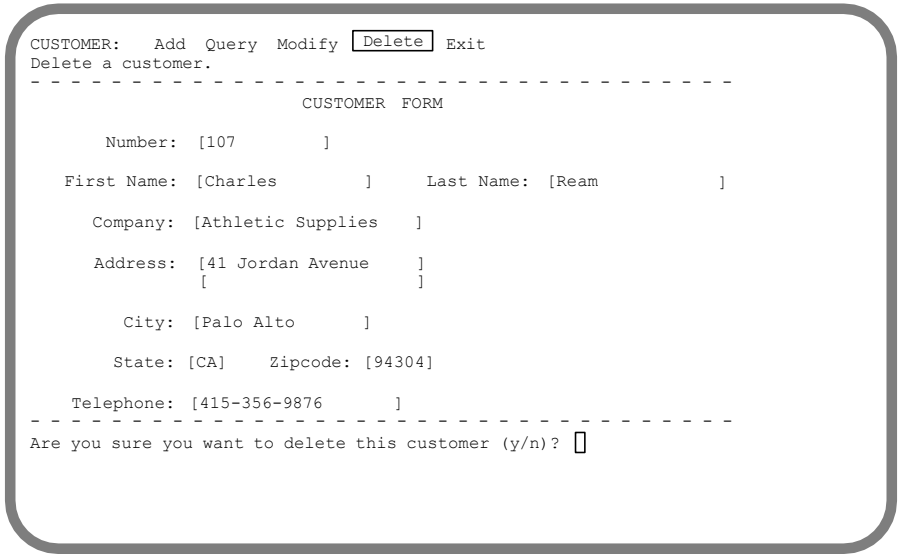
A problem occurs with the program if you choose the **Delete** option two or more times in succession.

To illustrate the undesirable result

1. Choose the **Query** option and enter search criteria for customer Charles Ream.
2. Enter `y` at the prompt to select this customer.
3. Choose the **Delete** option to remove this customer from the database.

   When you choose the **Delete** option, the **show_menu** function asks you to confirm that you want to delete the customer. The Application screen appears as in Figure 4-1.

Figure 4-1
Deleting a Customer



4. Enter `y` to confirm the deletion.

   When you make your entry, **show_menu** calls the **delete_row** function. This function deletes the customer from the database, clears the form, and displays the following message:

   ```
   Row deleted.
   ```

5. Choose the **Delete** option a second time.

   As soon as you select this option, the prompt reappears, even though no customer is currently displayed on the form, as shown in Figure 4-2.

Figure 4-2
Choosing the Delete Option a Second Time

```
CUSTOMER:   Add  Query  Modify  Delete  Exit
Delete a customer.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                          CUSTOMER  FORM

       Number:  [           ]

   First Name:  [               ]     Last Name:  [               ]

      Company:  [                    ]

      Address:  [                    ]
                [                    ]

         City:  [               ]

        State: [ ]     Zipcode: [      ]

    Telephone: [                  ]
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Are you sure you want to delete this customer (y/n)?
```

6. Enter `y` in response to this prompt.

   The message `"Row deleted."` redisplays, even though no customer is present on the form to delete.

7. Choose the **Delete** option and respond to the prompt one or more additional times to confirm that the message continues to display incorrectly.

Choosing Delete and Modify in Succession

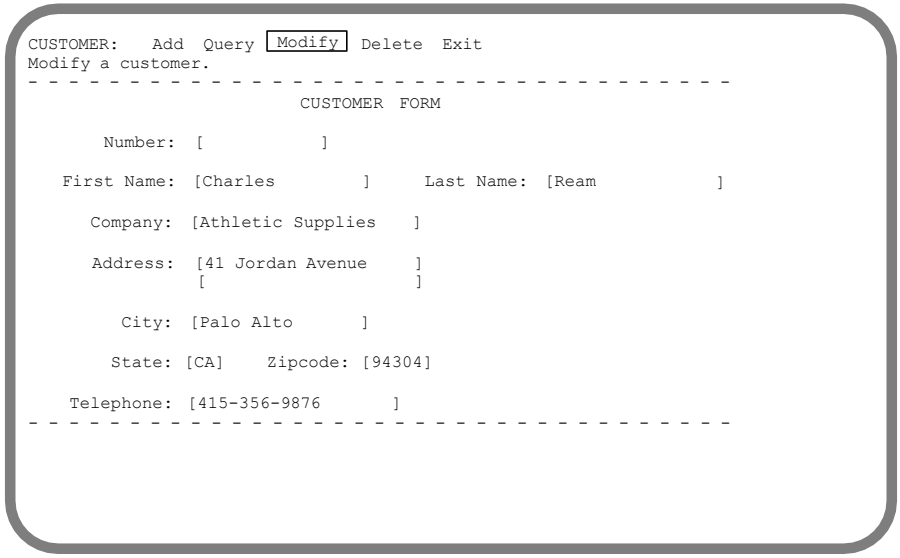
The second undesirable result occurs if you choose the **Modify** menu option immediately subsequent to carrying out the activities of the **Delete** option.

To illustrate the Modify bug

1. Choose the **Modify** option.

   As soon as you choose this option, the values for all but the **Number** field of deleted customer Charles Ream appear on the screen, and the program acts as if you can update a deleted record. The cursor moves to the **First Name** field on the form and waits for you to change the existing values, as shown in Figure 4-3.

Figure 4-3
Choosing the Delete and Modify Options in Sequence



2. Enter a sample value in the second **Address** field and press the **Accept** key (usually ESC) to record your changes.

   The message:

   ```
   Row updated.
   ```

   is displayed by the **change_data** function, and the cursor returns to the program menu.

3. Choose the **Query** option and enter search criteria for Charles Ream.

   The message:

   ```
   No customer rows found.
   ```

   is displayed, indicating that this customer is in fact no longer in the database.

Accessing the Debugger

You are now ready to use the Debugger to discover the cause of these problems.

To access the Debugger

1. Choose the **Exit** option to end the program and return to the Programmer's Environment.
2. Choose the **Debug** option from the **MODULE** menu.
3. Select the **customer** program.

The Restored Environment

When the Debugger windows appear on the terminal screen, the following information scrolls in the Command window:

```
Current search path:.
TERMINAL  DISPLAY  STATE
autotoggle            on
sourcetrace           on
displaystops          on
exitsource            on
printdelay            off
timedelay source      1
timedelay  command    2
source  lines         9
command  lines        10
(1)  trace  functions
          execute:  {  dump  all  }
```

These values, with the exception of SOURCE LINES and COMMAND LINES are restored from the **customer.4db** file that you created with the WRITE command at the end of Chapter 2, "Getting Started with the Debugger," and modified with the WRITE TRACE command at the end of Chapter 3, "Tracing Logic of the customer Program."

If you have not left the Programmer's Environment since carrying out the debugging steps in Chapter 2 and Chapter 3, these values are restored from a temporary file rather than from **customer.4db**. The same information is displayed whether the values are restored from a temporary file or from a **.4db** file. If you left the Programmer's Environment and did not enter the WRITE and WRITE TRACE commands at the end of Chapter 2 and Chapter 3, no values are saved or restored.

Reference Numbers

The specific reference numbers assigned by the Debugger to tracepoints and breakpoints are applicable only to the current session. Although the tracepoint currently displayed in the Command window, **trace functions { dump all }**, was the second one defined in the previous chapter, where it received a reference number of (2), it was the only one saved before exiting from the Debugger. This tracepoint is therefore restored in the current session with a reference number of (1).

Modifying the Environment

Before carrying out the activities in this chapter, you should reset the SOURCETRACE parameter to the default value of OFF. You can then observe more clearly the interaction of the program and the Debugger when breakpoints are reached.

Enter the following command to turn off the activity of the SOURCETRACE parameter:

```
turn off sourcetrace
```

Now, the Source window does not highlight each line of code as it executes.

The BREAK Command

The setting of breakpoints is a valuable debugging tool that allows you to suspend execution of a program and perform diagnostic tests. You can use the BREAK command to cause your program to suspend execution automatically in any of the following situations:

- A particular line of code executes.
- The value of a specific variable changes.
- A specific function executes.
- A particular expression evaluates to TRUE.

In addition, you can qualify a breakpoint in the following ways:

- Specify the number of times the breakpoint is reached before execution is suspended.
- Specify a condition that must be met before execution is suspended.

As with tracepoints, you can specify commands to execute when a breakpoint is reached by placing them in braces ({ }) following the breakpoint.

When the Debugger encounters a breakpoint, it highlights the next statement to be executed in the Source window. It also displays information in the Command window. This information includes the name of the function, the line number, and the name of the module at which the breakpoint was reached, as well as the output of any optional commands you have specified. The cursor remains in the Command window, and program execution does not resume until you issue a command such as CONTINUE.

The following sections illustrate the options for setting breakpoints, with examples from **customer.4gl**.

Breaking at a Line Number

The simple format for setting a breakpoint at a line number is as follows:

```
BREAK lineno
```

When you set a breakpoint at a line number, execution stops immediately prior to executing the line specified by the breakpoint. For example, to set a breakpoint at line number 251:

```
251 INPUT p_customer.fname THRU p_customer.phone
252     WITHOUT DEFAULTS FROM sc_cust.*
```

you would enter the following command:

```
break 251
```

This breakpoint causes program execution to be suspended immediately prior to executing the INPUT WITHOUT DEFAULTS statement.

Breaking at a Variable

The simple format for setting a breakpoint on a variable is as follows:

```
BREAK variable
```

To set a breakpoint on a local variable, you must either make its function the current function by displaying it in the Source window or qualify the variable name. For example, to set a breakpoint when the value of the local variable **exist** in the **query_data** function changes, you can enter the following command if **query_data** is the function currently displayed in the Source window:

```
break exist
```

This breakpoint causes program execution to stop whenever the value of **exist** changes from `0` (FALSE) to `1` (TRUE) or vice versa.

If **query_data** is not currently displayed in the Source window, you must qualify the variable by using either of the following conventions:

```
break (query_data) exist
```

or:

```
break  function.query_data.exist
```

You must qualify a global variable with the GLOBAL keyword when setting a breakpoint if there is a local variable with the same name in the function currently displayed in the Source window. See Chapter 9, "The Debugger Commands," for more information on specifying the scope of reference of variables.

Breaking at a Function

The simple format for setting a breakpoint at a function is as follows:

```
BREAK function
```

When you set a breakpoint at a function, execution is suspended as soon as the function is entered. The Source window highlights the first executable statement in the function. Do not use parentheses when setting a breakpoint at a function.

To set a breakpoint when the **change_data** function is called, enter the following command:

```
break change_data
```

This breakpoint causes program execution to stop when the user chooses the **Modify** option and the program calls the **change_data** function.

Breaking If an Expression Is True

The simple format for setting a breakpoint when a particular expression evaluates to TRUE is as follows:

```
BREAK IF condition
```

To set a breakpoint when the value of the local variable **exist** is set to TRUE, you can enter the following command when **query_data** is the function currently displayed in the Source window:

```
break if exist = TRUE
```

If **query_data** is not the current function, you must qualify the variable using the following convention:

```
break if function.query_data.exist = TRUE
```

This breakpoint causes program execution to be suspended whenever the value of **exist** is set to TRUE. You should compare this example with the one presented previously, in the section "Breaking at a Variable" on page 4-11. In the present example, program execution is suspended only if the value of **exist** evaluates to TRUE, rather than every time the value of **exist** changes.

Specifying a Count

You can qualify a breakpoint by specifying the number of times you want the condition specified in the breakpoint to occur before execution is suspended. Use a hyphen (-) followed by a number to indicate the number of times you want to reach the breakpoint before suspending execution.

When you specify a value for *count*, the Debugger decrements this value each time it encounters the conditions set forth in the breakpoint. Program execution stops when the value of *count* is `1`.

For example, to modify the breakpoint presented in the previous section so that execution is suspended the third time the value of **exist** evaluates to TRUE, you would enter the following command:

```
break -3 if exist = TRUE
```

Specifying a Condition

You can qualify a breakpoint set at a line number, variable, or function by specifying a condition that must be met for program execution to be suspended. For example, you can modify the breakpoint presented in the section "Breaking at a Function" on page 4-12:

```
break change_data
```

so that program execution is suspended only if the value of the global variable **p_customer.customer_num** is greater than 110. To do so, you can enter the following command:

```
break change_data if p_customer.customer_num > 110
```

This breakpoint suspends program execution if you choose to modify the values of a previously selected customer whose customer number is greater than 110.

Setting the First Breakpoint for the Current Session

You have seen that undesirable results occur in the **customer** program when the **Delete** menu option is chosen two or more times in succession. The first evidence that the program is working improperly occurs when the prompt:

```
Are you sure you want to delete this customer (y/n)?
```

appears the second time the **Delete** option is chosen in sequence, even though no customer is currently displayed on the form. One way to approach the task of debugging the program is to set a breakpoint right before this statement executes so that you can perform some diagnostic tests. The statement that occurs immediately prior to this prompt is line 91 in the **show_menu** function:

```
91   IF chosen THEN
```

Setting a breakpoint at line 91 causes program execution to stop right before line 91 is executed, and gives you the opportunity to experiment with a different value for **chosen**.

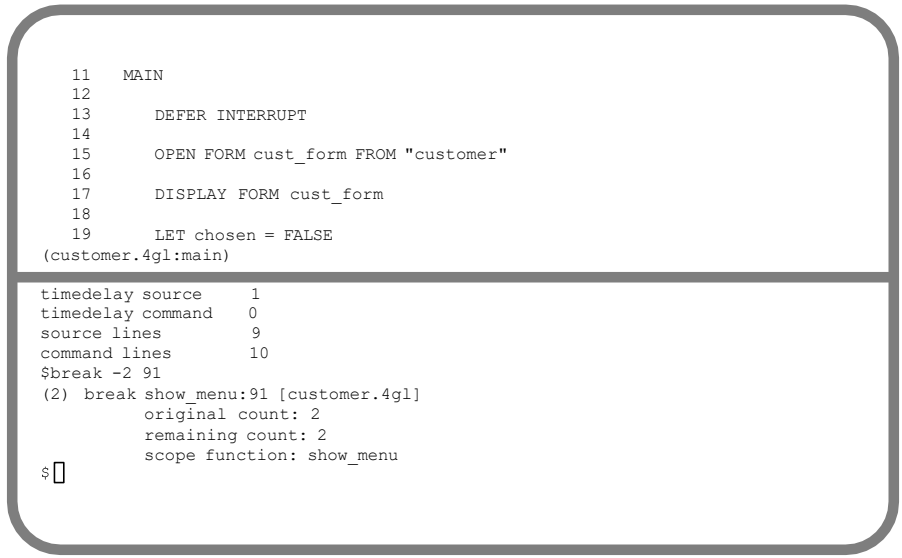
Because the problem becomes apparent the second time **Delete** is chosen in sequence, you can use the *count* option to cause program execution to stop the second time the program prepares to execute line 91.

Enter the following command to suspend execution of the program the second time the program prepares to execute line 91:

```
break -2 91
```

Figure 4-4 illustrates the setting of this breakpoint and the Debugger response.

Figure 4-4
Setting the First Breakpoint



When you set a breakpoint, the Debugger assigns it a reference number and displays this number in the Command window. Both tracepoints and breakpoints are assigned the next sequential number, and no distinction is made between them. In this example, the breakpoint is assigned the reference number (2) because there is an existing tracepoint with the number (1).

When you set or LIST a breakpoint that includes the *count* option, the Debugger displays both the original number you have specified and the number of cycles remaining before the breakpoint executes. Because you have not yet begun program execution with the breakpoint, the two numbers are identical.

The DISABLE Command

Before running the program with the breakpoint you have set, you should disable, or turn off the activity of, the tracepoint. This allows you to observe more clearly the interaction of the program and the Debugger when breakpoints are reached.

The DISABLE command allows you to turn off the activity specified by a tracepoint or breakpoint without removing the point itself. A disabled tracepoint or breakpoint remains defined and can be enabled, or reactivated, at a later time.

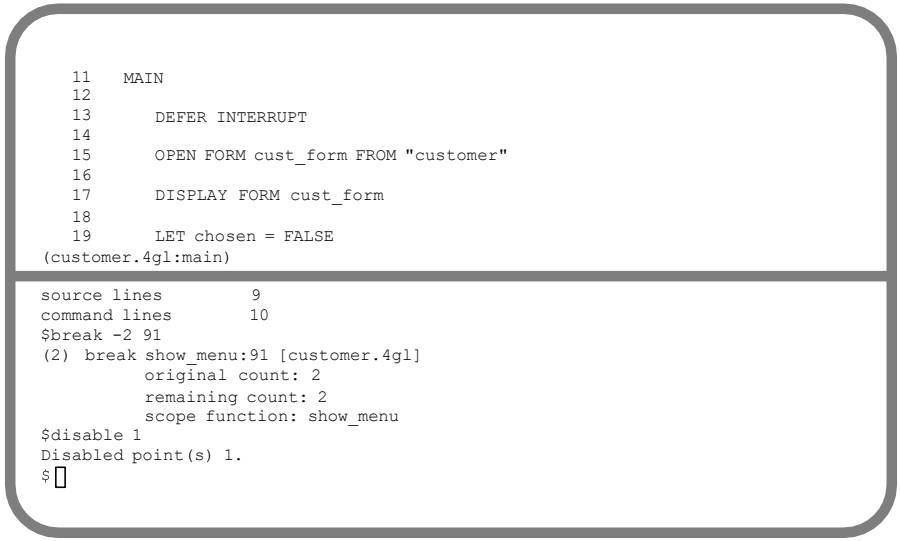
You can refer to tracepoints or breakpoints by their reference numbers when disabling them.

For example, you can enter the following command to disable the current tracepoint, **trace functions { dump all }**:

```
disable  1
```

The Debugger confirms that the tracepoint is disabled by echoing this information in the Command window. Figure 4-5 illustrates your command to disable the tracepoint and the Debugger response.

Figure 4-5
Disabling a Tracepoint



Now, when you run the **customer** program, the Debugger does not trace functions as they execute or dump the values of variables as the functions change.

Reaching the First Breakpoint

You are now ready to run the program with the breakpoint you have set.

To run the program with breakpoints

1. Enter the RUN command or press F5 to start the Debugger.

   The **PROGRAM** menu appears.

2. Choose the **Query** option, and enter the search string `"*Sports*"` in the **Company** field.

3. Press RETURN in response to the prompt:

   ```
   Enter 'y' to select this customer or RETURN to view next customer:
   ```

   until the information for customer Carole Sadler of Sports Spot is displayed.

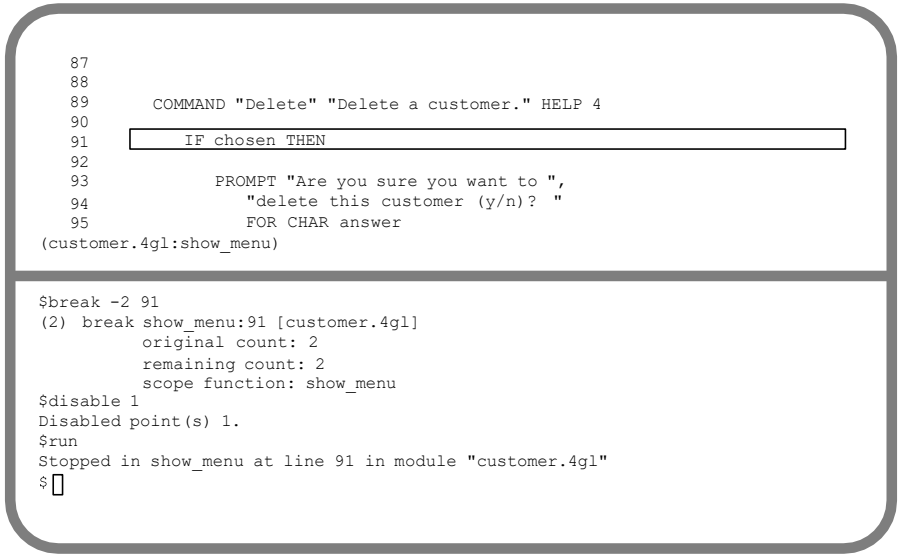
4. Enter y to select this customer.
5. Choose the **Delete** option to remove this customer from the database.
6. Enter y in response to the prompt:

```
Are you sure you want to delete this customer (y/n)?
```

   The program executes line 91 right before it displays the second prompt. Because the breakpoint you have set specifies that execution is to stop immediately prior to executing this line for the second time, there is no interruption at this point. The cursor returns to the **PROGRAM** menu and waits for you to make another selection.
7. Choose the **Delete** option again.

   Now the program prepares to execute line 91 a second time, and the condition specified in the breakpoint is met. Program execution is suspended. The Debugger windows redisplay is shown in Figure 4-6.

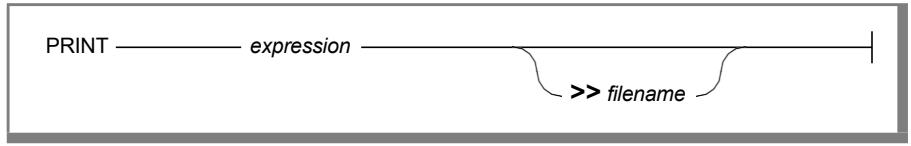
Figure 4-6
Reaching the First Breakpoint



The Debugger highlights line 91 in the Source window. This is the next statement to execute when operation resumes.

The Command window displays the function name, line number, and module name at which the breakpoint occurred. The cursor is at the $ prompt, and you can enter any valid Debugger command.

The PRINT Command



The DUMP command with which you are already familiar prints the value of local, global, and module variables in the currently executing function. The PRINT command prints the value of an individual variable or expression in an active function.

The PRINT command outputs the following information to the Command window or, optionally, to a file:

- The value of a simple program variable
- The value of each member of a program record
- The value of each member of a program array
- The value of an arithmetic expression

To print the value of a variable, the function in which the variable receives its value must be active. A function is active as long as it, or any functions called by it, are executing.

The following sections illustrate printing the value of a variable and printing the values of the members of a program record, with examples from the **customer** program. Chapter 6, "Tracing Logic of the cust_order Program," provides additional examples of the PRINT command, including the use of PRINT to display the elements of a program array.

Printing the Value of a Variable

To print the value of the local variable **answer** in the **show_menu** function when **show_menu** is the function displayed in the Source window, enter the following command:

```
print answer
```

If **show_menu** is not currently displayed in the Source window, you need to qualify the variable name using the following convention:

```
print   function.show_menu.answer
```

If the value of **answer** is y, the output of the PRINT command is as follows:

```
           remaining count: 2
           scope function: show_menu
$disable 1
Disabled point(s) 1.
$run
Stopped in show_menu at line 91 in module "customer.4gl"
$print answer
customer.4gl:show_menu.answer    = "y"
$
```

Printing the Values of Record Members

To print the values of the members of the global record **p_customer**, enter the following command:

```
print p_customer
```

If the active customer is Arnold Sipes, the following output of the PRINT command scrolls through the Command window:

```
global:p_customer  = record
    customer_num = 117
    fname = "Arnold          "
    lname = "Sipes           "
    company = "Kids Korner            "
    address1 = "850 Lytton Court       "
    address2 = (null)
    city = "Redwood City     "
    state = "CA"
    zipcode = "94063"
    phone = "415-245-4578       "
end record
```

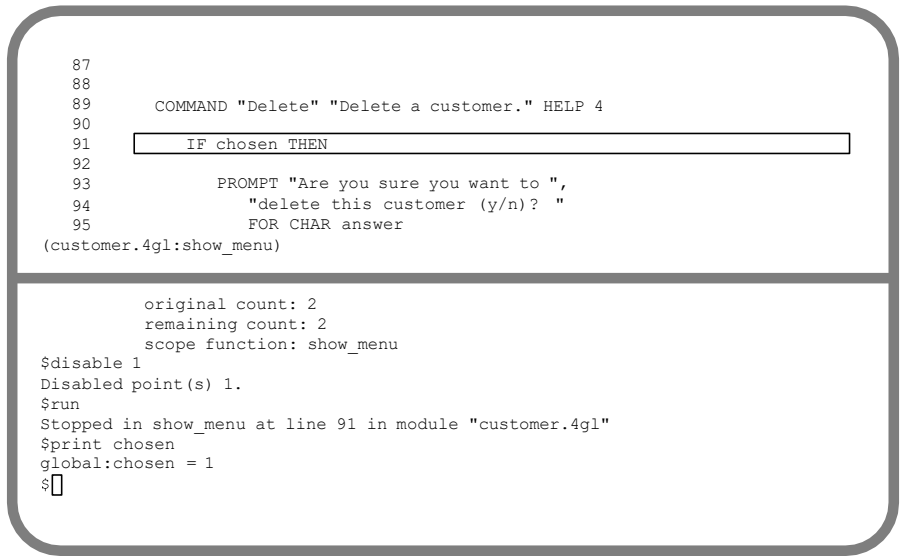
Entering the PRINT Command

Enter the PRINT command as follows to display the current value of **chosen**:

```
print  chosen
```

The output of the PRINT command indicates that the current value of **chosen** is `1`, or TRUE, as shown in Figure 4-7.

Figure 4-7
Printing the Value of chosen



The LET Command

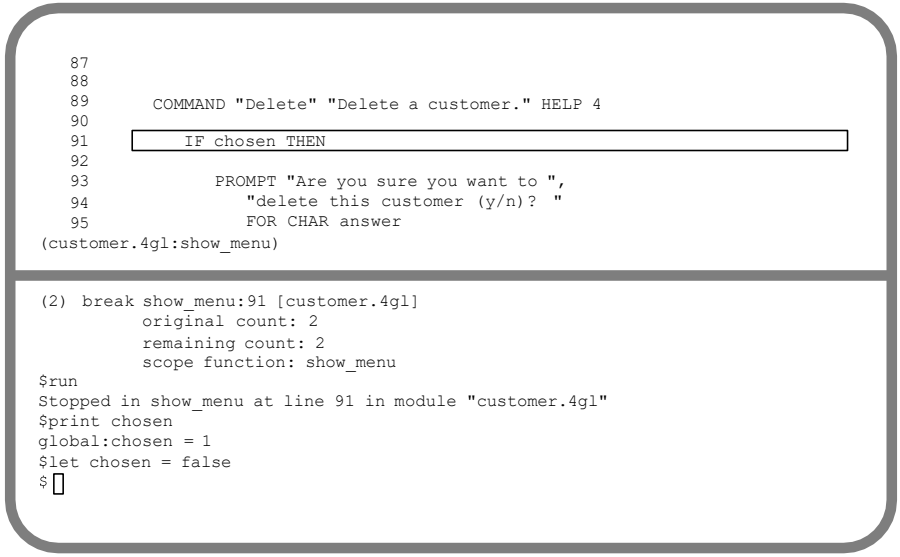
You can use the LET command as you would in 4GL to assign a value to a variable. Using the LET command in the Debugger allows you to change the value of a variable while operation is suspended and to continue running the program with the new value.

In the present example, you have seen that a prompt displays incorrectly if you choose the **Delete** option twice in succession. You know also that this prompt displays when the value of the variable **chosen** is TRUE. The variable **chosen** is a flag with the possible values of TRUE or FALSE. A logical debugging step, therefore, is to change the value of **chosen** to FALSE and observe the behavior of the program with the new value.

Enter the LET command as follows to change the value of **chosen** to FALSE:

```
let chosen = false
```

Figure 4-8
Changing the Value of chosen



The STEP Command

At this point, you could resume program execution using the CONTINUE command. Entering CONTINUE would cause the Debugger to resume executing the program with the value of **chosen** set to FALSE and would allow you to observe the functioning of the program with the new value.

An alternative approach is to use the STEP command to execute one or more individual 4GL statements. The simple format of the STEP command is as follows:

```
STEP n
```

where n specifies the number of statements you want to execute. If no value is specified for *n*, the Debugger executes the next statement.

To execute the next statement, you can either enter:

```
step
```

or press F2. To execute the next five statements, you would enter:

```
step 5
```

When you enter the STEP command, the Debugger executes the statement or series of statements you have specified, and then returns control to you. The Command window displays the function name, the line number, and the name of the module at which execution was suspended following the step.

Chapter 6 provides additional examples of the STEP command. See Chapter 9 for a complete description of the STEP command and its options.

Entering the STEP Command

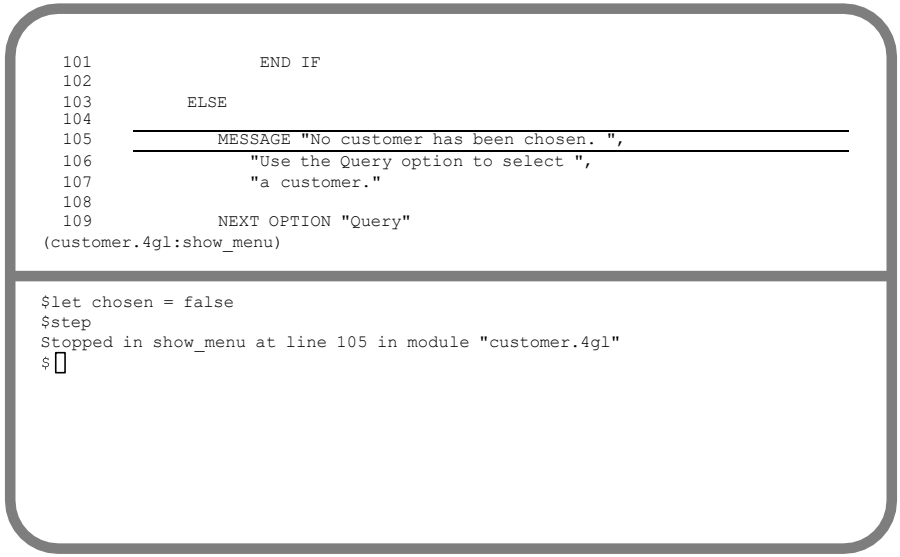
It is convenient to use STEP in the present example because you are interested in observing the next statement that the Debugger prepares to execute after the value of **chosen** in line 91 is changed to FALSE.

Enter the STEP command or press F2 to tell the Debugger to execute the statement currently highlighted in the Command window:

```
91   IF chosen THEN
```

Figure 4-9 illustrates the appearance of the Debugger windows immediately following execution of this command.

Figure 4-9
Output from the STEP command



The Debugger now highlights the next statement to be executed in the Source window. This statement is line 105 in the **show_menu** function. It displays the message that no customer has been chosen and instructs you to use the **Query** option to select one. This is the message that the program displays automatically the first time the user attempts to use the **Delete** option without previously selecting a customer. The Command window displays the information that execution stopped at this statement.

This series of debugging steps suggests that the problems with **customer.4gl** can be corrected by setting the value of **chosen** to FALSE upon return from the **delete_row** function. If the value of **chosen** is set to FALSE at this point in the program, the ELSE condition following line 103 executes whenever you have failed to select a customer instead of only the first time.

The NOBREAK Command

You can use the NOBREAK command to permanently remove a breakpoint. In the present example, it is desirable to remove the current breakpoint, **break -2 91**, before continuing with the debugging steps in this chapter. Because the current value of *count* is 1, this breakpoint now suspends execution every time the program prepares to execute line 91 unless removed or disabled.

You can refer to breakpoints by their reference numbers when removing them. For example, you can enter the following command to remove the breakpoint with the reference number (2), **break -2 91**:

```
nobreak 2
```

Now the Debugger does not suspend program execution immediately prior to executing line 91.

Setting the Second Breakpoint for the Current Session

You can determine that failing to reset **chosen** to FALSE following the return from the **delete_row** function is responsible for the second problem detected with the program. You recall that choosing the **Modify** option immediately after a deletion causes the values **First Name** field through **Phone** field for the deleted customer to appear on the screen. It appears that you can modify the deleted row. Because the program calls the **change_data** function under the same condition that it calls **delete_row** (whenever the value of **chosen** is TRUE), you can use a similar series of debugging steps as in the previous example. Specifically, the debugging strategy is as follows:

1. Set a breakpoint at line 74 in the **show_menu** function:

   ```
   74 IF chosen THEN
   ```

2. Use the LET command to reset the value of **chosen** to FALSE.
3. Use the STEP command to execute line 74 with the new value.

Because you are reasonably sure of the steps you want to take when the breakpoint is reached, you can save time by combining these commands. Specifically, you can specify LET and STEP between the curly braces as commands to execute when the breakpoint is reached. Enter the second breakpoint as follows:

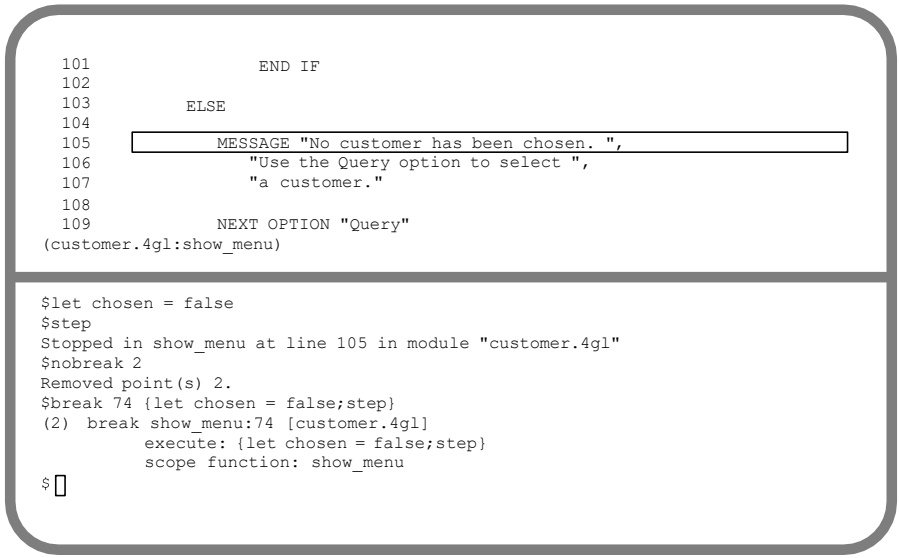
```
break 74 {let chosen = false;step}
```

When you use the braces to specify multiple commands for the Debugger to execute upon reaching a breakpoint, the commands must be separated with semicolons. If your commands span more than one line, you can continue making your entry on additional lines. A **>** prompt appears if you press RETURN before terminating your entry with the right brace symbol (}).

The Debugger does not execute a RUN, CONTINUE, or STEP command until it has processed all other commands in the sequence. Other commands execute in the order in which they appear.

Figure 4-10 illustrates your setting of the second breakpoint and the Debugger response.

Figure 4-10
Specifying Commands with a Breakpoint



Observe that the Debugger assigns the new breakpoint a reference number of (2) because you have removed the previous point with this number.

Reaching the Second Breakpoint

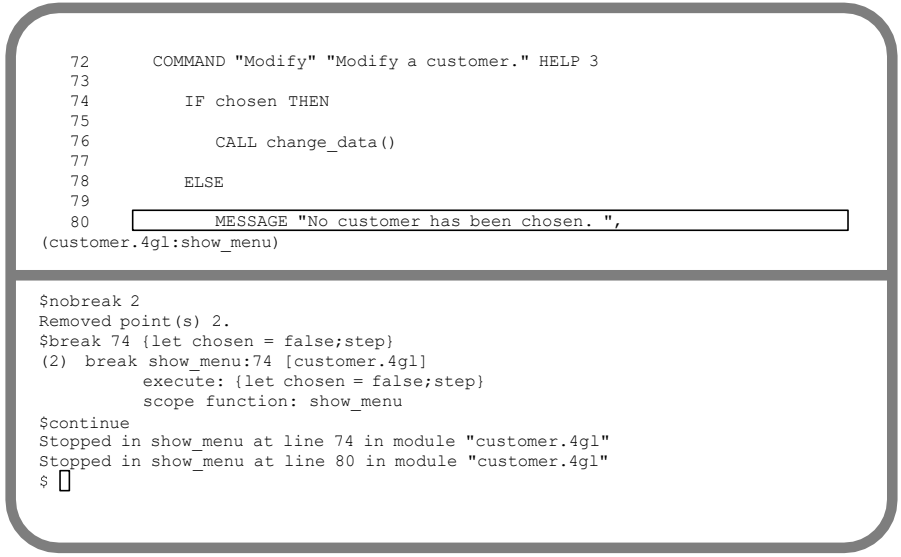
The following procedure describes how to resume execution of the program and observe the activity of the second breakpoint.

To reach the second breakpoint

1. Enter the CONTINUE command or press F4.
2. Choose the **Query** option, enter the search criterion for customers with zip code `"94062"` and press ESC.
3. Press RETURN, if necessary, in response to the prompt until the record for customer Roy Jaeger is displayed.
4. Enter `y` to select this customer.
5. Choose the **Delete** option to remove this customer from the database.
6. Enter `y` in response to the prompt to carry out the deletion.
7. Choose the **Modify** option.

As soon as you choose the **Modify** option, the program prepares to execute line 74. Because you have set a breakpoint at this line, program execution is suspended. The Debugger then automatically resets the value of **chosen** and steps to the next executable statement with the new value. Figure 4-11 illustrates the appearance of the Debugger windows after the breakpoint has been reached and after both the LET and the STEP statements have executed.

Figure 4-11
Reaching the Second Breakpoint



The Source window highlights the next statement to be executed after you have stepped through line 74. You observe that this is line 80, which correctly displays the message that no customer has been selected.

The Debugger generates two lines of output to the Command window. The first line:

```
Stopped in show_menu at line 74 in module "customer.4gl"
```

was generated by the BREAK command when the breakpoint was reached.

The second line:

```
Stopped in show_menu at line 80 in module "customer.4gl"
```

was generated by the STEP command after you used it to execute line 74.

You have learned from these debugging steps, therefore, that the value of **chosen** must be reset to FALSE following the return from **delete_row**. "Correcting the customer Program" on page 4-31 provides instructions for making the necessary change.

Saving and Exiting

This chapter concludes the examples based on the program **customer.4gl.** In the course of working with the program, you have created and modified the contents of a file called **customer.4db**, which the Debugger uses to restore the debugging environment for this program. The contents of this file are currently as follows:

```
alias f1 = help
alias f2 = step
alias f3 = step into
alias f4 = continue
alias f5 = run
alias f6 = list break trace
alias f7 = list
alias  f8  =  dump
alias  f9  =  exit
use = .
turn on autotoggle
turn on sourcetrace
turn on displaystops
turn on exitsource
turn off printdelay
timedelay source 1
timedelay command 0
list display
trace  functions  {  dump  all  }
```

You observe that SOURCETRACE is saved in this file with a value of ON, even though you have turned SOURCETRACE OFF for the current session. To save the current value of SOURCETRACE, you would need to issue the WRITE command with the DISPLAY option. In this example, WRITE DISPLAY would append the current values of all the terminal display parameters to the **customer.4db** file. To save the current breakpoint as part of this file, you can use the WRITE command with the option BREAK.

To save the breakpoint as part of the file

1. Enter the following command to save the current breakpoint, **break 74 {let chosen = false;step}**:

```
write break
```

   The contents of the **customer.4db** file are modified as follows:

```
alias f1 = help
alias f2 = step
alias f3 = step into
alias f4 = continue
alias f5 = run
alias f6 = list break trace
alias f7 = list
alias  f8  =  dump
alias  f9  =  exit
use = .
turn on autotoggle
turn on sourcetrace
turn on displaystops
turn on exitsource
turn off printdelay
timedelay source 1
timedelay command 0
list display
trace  functions  {  dump  all  }
break  74  {  let  chosen  =  false  ;  step  }
```

2. Enter the EXIT command or press F9 to exit from the Debugger and return to the Programmer's Environment.

Correcting the customer Program

You have determined that in order for the program to work correctly, the value of the **chosen** variable needs to be reset to FALSE following the return from **delete_row**. The proper placement of this statement is in the **show_menu** function immediately following line 99:

```
99   CALL delete_row()
```

To make this correction to the program from the Programmer's Environment

1. Choose the **Modify** option from the **MODULE** menu.
2. Select the **customer** program.
3. Use the system editor to move to line 99 of this module:

   ```
   99   CALL delete_row()
   ```

4. Insert the following line immediately after line 99:

   ```
   LET chosen = FALSE
   ```

5. Save the file using the command appropriate to your system editor.
6. Choose the **Compile** option from the **MODIFY MODULE** menu.
7. Choose the **Runable** option from the **COMPILE MODULE** menu.

   You should see the following message when compilation is complete:

   ```
   A module was successfully compiled.
   ```

   If errors are discovered during the compilation, you should return to the **MODULE** menu and choose the **Modify** option to correct the module.

Following is a complete listing of the **show_menu** function with this correction:

```
FUNCTION show_menu()

   DEFINE answer  CHAR(1)

   MESSAGE "Type the first letter of the option ",
      "you want to select or CONTROL I for Help."

   MENU "CUSTOMER"
```

```
COMMAND "Add" "Add a new customer." HELP 1

   LET answer = "y"

      WHILE answer = "y"

         CALL enter_row()

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

      END WHILE

   CLEAR FORM

COMMAND "Query" "Search for a customer." HELP 2

   CALL query_data()

   IF chosen THEN

      NEXT OPTION "Modify"

   END IF

COMMAND "Modify" "Modify a customer." HELP 3

   IF chosen THEN

      CALL change_data()

   ELSE

      MESSAGE "No customer has been chosen. ",
         "Use the Query option to select ",
         "a customer."

      NEXT OPTION "Query"

   END IF


COMMAND "Delete" "Delete a customer." HELP 4

   IF chosen THEN

      PROMPT "Are you sure you want to ",
         "delete this customer (y/n)? "
         FOR CHAR answer

         IF answer = "y" THEN

            CALL delete_row()
```

```
                LET chosen = FALSE

             END IF

         ELSE

            MESSAGE "No customer has been chosen. ",
               "Use the Query option to select ",
               "a customer."

            NEXT OPTION "Query"

         END IF


      COMMAND "Exit" "Leave the CUSTOMER menu." HELP 5

         EXIT MENU

      END MENU

   END FUNCTION
```

A breakpoint is a valuable debugging tool that allows you to suspend operation of your program under preset conditions. These conditions include when a particular statement or function executes, or when the value of a variable changes. The Source window highlights the next statement to execute when operation resumes. The Command window records the function name, line number, and module name where the breakpoint occurred.

While execution is suspended, you can perform diagnostic tests such as printing the current values of variables or changing the value of a variable and stepping to the next statement with the new value.

Tracepoints and breakpoints can be disabled, or deactivated, for a particular series of steps, or they can be removed entirely.

Chapter

5

A Multi-Module Program: cust_order

In This Chapter

You are now familiar with the basic operation of the Debugger and with the principal tools available to you for tracing program logic and for verifying the operation of INFORMIX-4GL programs. This chapter introduces a program of greater complexity than the one used in the previous examples and summarizes the points that you need to be aware of when working with multi-module programs.

When compiled and run, the sample program produces two common runtime errors. Chapter 6, "Tracing Logic of the cust_order Program," introduces additional commands and capabilities of the Debugger, using aspects of the program that are working correctly. Chapter 7, "Analyzing Runtime Errors in the cust_order Program," illustrates the use of the Debugger to diagnose the cause of fatal errors and provides instructions for correcting them.

This chapter uses examples based on the **cust_order.4gi** program that is produced when you compile the following modules:

- **globals.4gl**
- **main.4gl**
- **order.4gl**

This chapter assumes that you are familiar with more advanced 4GL statement syntax, including the INPUT ARRAY and DISPLAY ARRAY statements, the window management statements, and the use of scrolling cursors. The following topics are covered in this chapter:

- How to define and compile the **cust_order** program
- How to use the VIEW command to display a program module in the Source window
- How to set a tracepoint or breakpoint at a line number in a module
- How to set a tracepoint or breakpoint on a module variable

The examples in this chapter are based on the following modules and forms provided with the demonstration database:

- **globals.4gl**
- **main.4gl**
- **order.4gl**
- **orderform.per**
- **stock_sel.per**

The cust_order Program

In "Defining and Compiling the Program" on page 5-15, you use the Programmer's Environment to define and compile the **cust_order** program. When compiled, this program allows the user to place an order for a customer and to enter up to 10 items in the order. A second activity, that of finding and displaying existing orders, is anticipated by the program but is not currently implemented.

The purpose of the **cust_order** program is to give you additional practice with the Debugger using a program of moderate complexity. In an application program designed for actual use, you would probably want to implement **Find-Order** and perhaps other options, and provide more thorough checking for data integrity.

The three component modules of the **cust_order** program appear on the following pages and are followed by brief explanatory notes on each module. These modules are similar, though not identical, to several of the modules that make up the **d4_demo.4gi** program provided with the Rapid Development System demonstration database.

A complete description of the functions defined in these modules is provided in Appendix C, "Sample Programs." Consult Appendix C if you would like more information on the program after studying the example.

MODULE #1: globals.4gl

The following code constitutes the **globals.4gl** module:

```
 1 DATABASE stores7
 2
 3 GLOBALS
 4    DEFINE
 5       p_customer RECORD LIKE customer.*,
 6       p_orders RECORD
 7          order_num LIKE orders.order_num,
 8          order_date LIKE orders.order_date,
 9          po_num LIKE orders.po_num,
10          ship_instruct LIKE orders.ship_instruct
11       END RECORD,
12       p_items ARRAY[10] OF RECORD
13          item_num LIKE items.item_num,
14          stock_num LIKE items.stock_num,
15          manu_code LIKE items.manu_code,
16          description LIKE stock.description,
17          quantity LIKE items.quantity,
18          unit_price LIKE stock.unit_price,
19          total_price LIKE items.total_price
20       END RECORD,
21       p_stock ARRAY[15] OF RECORD
22          stock_num LIKE stock.stock_num,
23          manu_code LIKE manufact.manu_code,
24          manu_name LIKE manufact.manu_name,
25          description LIKE stock.description,
26          unit_price LIKE stock.unit_price,
27          unit_descr LIKE stock.unit_descr
28       END RECORD,
29       stock_cnt INTEGER
30 END GLOBALS
```

The globals file defines two program records, two program arrays, and a variable to serve as a counter:

- The **p_customer** record is defined with variables corresponding to all the columns of the **customer** table.
- The **p_orders** record is defined with variables corresponding to four of the columns of the **orders** table.
- The **p_items** program array is defined as an array of 10 records with variables corresponding to five of the columns of the **items** table and two of the columns of the **stock** table.
- The **p_stock** program array is defined as an array of 15 records with variables corresponding to four of the columns of the **stock** table and the two columns of the **manufact** table.
- The **stock_cnt** variable is a counter that keeps track of the number of rows retrieved from the database into the **p_stock** array.

MODULE #2: main.4gl

The following code constitutes the **main.4gl** module:

```
 1  GLOBALS
 2   "globals.4gl"
 3
 4
 5  MAIN
 6
 7     DEFER INTERRUPT
 8
 9     OPEN FORM order_form FROM "orderform"
10     DISPLAY FORM order_form
11        ATTRIBUTE(MAGENTA)
12     MENU "ORDERS"
13        COMMAND "Add-order"
14        "Enter new order to database"
15           CALL add_order()
16        COMMAND "Find-order" "Look up and display orders"
17           CALL find_order()
18        COMMAND "Exit" "Exit program and return to operating system"
19           CLEAR SCREEN
20           EXIT PROGRAM
21     END MENU
22
23  END MAIN
24
25
26
27  FUNCTION mess(str, mrow)
28     DEFINE str CHAR(80),
29        mrow SMALLINT
```

```
30
31      DISPLAY " ", str CLIPPED AT mrow,1
32      SLEEP 3
33      DISPLAY "" AT mrow,1
34  END FUNCTION
35
36
37  FUNCTION clear_menu()
38
39      DISPLAY "" AT 1,1
40      DISPLAY "" AT 2,1
41  END FUNCTION
42
43
44
45
46  FUNCTION fetch_stock()
47
48      DECLARE stock_list CURSOR FOR
49      SELECT stock_num, manufact.manu_code,
50         manu_name, description, unit_price, unit_descr
51         FROM stock, manufact
52         WHERE stock.manu_code = manufact.manu_code
53         ORDER BY stock_num
54      LET stock_cnt = 1
55      FOREACH stock_list INTO p_stock[stock_cnt].*
56         LET stock_cnt = stock_cnt + 1
57      END FOREACH
58      LET stock_cnt = stock_cnt - 1
59  END FUNCTION
60
61  FUNCTION query_customer()
62      DEFINE where_part CHAR(200),
63         query_text CHAR(250),
64         answer CHAR(1),
65         mrow, chosen, exist SMALLINT
66
67      CLEAR FORM
68      CALL clear_menu()
69
70      MESSAGE "Enter criteria for selection"
71      CONSTRUCT where_part ON customer.* FROM customer.*
72      MESSAGE ""
73      IF int_flag THEN
74         LET int_flag = FALSE
75         CLEAR FORM
76         ERROR "Customer query aborted" ATTRIBUTE(RED, REVERSE)
77         LET p_customer.customer_num = NULL
78         RETURN (p_customer.customer_num)
79      END IF
80      LET query_text = "select * from customer where ", where_part CLIPPED,
81         " order by lname"
82      PREPARE statement_1 FROM query_text
83      DECLARE customer_set SCROLL CURSOR FOR statement_1
84
85      OPEN customer_set
86      FETCH FIRST customer_set INTO p_customer.*
87      IF status = NOTFOUND THEN
88         LET exist = FALSE
89      ELSE
90         LET exist = TRUE
```

```
 91        DISPLAY BY NAME p_customer.* ATTRIBUTE(MAGENTA)
 92        MENU "BROWSE"
 93           COMMAND "Next" "View the next customer in the list"
 94              FETCH NEXT customer_set INTO p_customer.*
 95                 IF status = NOTFOUND THEN
 96                    ERROR "No more customers in this direction"
 97                       ATTRIBUTE(RED, REVERSE)
 98                    FETCH LAST customer_set INTO p_customer.*
 99                 END IF
100              DISPLAY BY NAME p_customer.* ATTRIBUTE(MAGENTA)
101           COMMAND "Previous" "View the previous customer in the list"
102              FETCH PREVIOUS customer_set INTO p_customer.*
103                 IF status = NOTFOUND THEN
104                    ERROR "No more customers in this direction"
105                       ATTRIBUTE(RED, REVERSE)
106                    FETCH FIRST customer_set INTO p_customer.*
107                 END IF
108              DISPLAY BY NAME p_customer.* ATTRIBUTE(MAGENTA)
109           COMMAND "First" "View the first customer in the list"
110              FETCH FIRST customer_set INTO p_customer.*
111              DISPLAY BY NAME p_customer.* ATTRIBUTE(MAGENTA)
112           COMMAND "Last" "View the last customer in the list"
113              FETCH LAST customer_set INTO p_customer.*
114              DISPLAY BY NAME p_customer.* ATTRIBUTE(MAGENTA)
115           COMMAND "Select" "Exit BROWSE selecting the current customer"
116              LET chosen = TRUE
117              EXIT MENU
118           COMMAND "Quit" "Quit BROWSE without selecting a customer"
119              LET chosen = FALSE
120              EXIT MENU
121        END MENU
122     END IF
123     CLOSE customer_set
124
125     CALL clear_menu()
126     IF NOT exist THEN
127        CLEAR FORM
128        CALL mess("No customer satisfies query", mrow)
129        LET p_customer.customer_num = NULL
130        RETURN (FALSE)
131     END IF
132     IF NOT chosen THEN
133        CLEAR FORM
134        CALL mess("No selection made", mrow)
135        LET p_customer.customer_num = NULL
136        RETURN (FALSE)
137     END IF
138     RETURN (TRUE)
139  END FUNCTION
```

The main module references the **globals** file with the GLOBALS statement. It consists of the MAIN block and of four functions:

- The MAIN section issues the DEFER INTERRUPT command, opens and displays the **orderform** screen form, and presents the program menu. The user can choose to add a new order for a customer or to look up existing orders. The second of these activities is anticipated but not yet implemented.
- The **mess** function displays a character string at a particular row in the first column of the screen. It receives the character string and the number of the row as parameters.
- The **clear_menu** function uses DISPLAY "" statements to clear the first two lines of the terminal screen.
- The **fetch_stock** function declares the cursor **stock_list** and retrieves rows into the **p_stock** program array by means of a FOREACH loop. It uses the global variable **stock_cnt** as a counter for the rows retrieved.
- The **query_customer** function uses a CONSTRUCT statement to perform a query by example and declares the scrolling cursor **customer_set**. It then opens this cursor and fetches the resulting rows into the **p_customer** program record. The function displays a menu that allows you to select the next, previous, first, or last customer in the active set. It returns a value of FALSE (0) to the calling function if no customer is found or selected and a value of TRUE (1) if a customer is both found and selected. If the user presses the **Interrupt** key during the CONSTRUCT statement, the function sets the value of the **_p_customer.customer_num** to NULL and returns this value to the calling function.

MODULE #3: order.4gl

The following code constitutes the **order.4gl** module:

```
 1  GLOBALS
 2     "globals.4gl"
 3
 4     DEFINE query_stat INTEGER
 5
 6
 7  FUNCTION add_order()
 8     DEFINE pa_curr, s_curr INTEGER
 9
10     LET query_stat = query_customer()
11     IF query_stat IS NULL OR query_stat = 0 THEN
12       RETURN
13     END IF
14     DISPLAY by name p_customer.* ATTRIBUTE(CYAN)
15
16     MESSAGE "Enter the order date, PO number and shipping instructions."
17     INPUT BY NAME p_orders.order_date, p_orders.po_num,
18        p_orders.ship_instruct
19     IF int_flag THEN
20        LET int_flag = FALSE
21        CLEAR FORM
22        ERROR "Order input aborted" ATTRIBUTE (RED, REVERSE)
23        RETURN
24     END IF
25     INPUT ARRAY p_items FROM s_items.*
26        BEFORE FIELD stock_num
27          MESSAGE "Press ESC to write order"
28          DISPLAY "Enter a stock number or press CTRL-B to scan stock list"
29              AT 1,1
30        BEFORE FIELD manu_code
31           MESSAGE "Press ESC to write order"
32           DISPLAY "" AT 1, 1
33           DISPLAY "Enter a manufacturer code or press CTRL-B to scan ",
34              "stock list" at 1, 1
35
36        BEFORE FIELD quantity
37           MESSAGE "Press ESC to write order"
38           DISPLAY "" AT 1,1
39           DISPLAY  "Enter the item quantity" AT 1, 1
40           ON KEY (CONTROL-B)
41              IF INFIELD(stock_num) OR INFIELD(manu_code) THEN
42                 LET pa_curr = arr_curr()
43                 LET s_curr = scr_line()
44                 CALL fetch_stock()
45                 CALL get_stock() RETURNING
46                   p_items[pa_curr].stock_num,  p_items[pa_curr].manu_code,
47                   p_items[pa_curr].description,  p_items[pa_curr].unit_price
48                 DISPLAY p_items[pa_curr].stock_num
49                    TO s_items[s_curr].stock_num
50                 DISPLAY p_items[pa_curr].manu_code
51                    TO s_items[s_curr].manu_code
52                 DISPLAY p_items[pa_curr].description
53                    TO s_items[s_curr].description
54                 DISPLAY p_items[pa_curr].unit_price
55                    TO s_items[s_curr].unit_price
56                 NEXT FIELD quantity
```

```
57              END IF
58          AFTER FIELD stock_num, manu_code
59             LET pa_curr = arr_curr()
60             IF p_items[pa_curr].stock_num IS NOT NULL
61                AND p_items[pa_curr].manu_code IS NOT NULL
62             THEN
63                CALL get_item()
64             END IF
65
66          AFTER FIELD quantity
67             MESSAGE ""
68             LET pa_curr = arr_curr()
69             IF p_items[pa_curr].unit_price IS NOT NULL
70                AND p_items[pa_curr].quantity IS NOT NULL
71             THEN
72                CALL item_total()
73             ELSE
74                ERROR "A valid stock code, manufacturer, and ",
75                "quantity must all be entered" ATTRIBUTE (RED, REVERSE)
76                NEXT FIELD stock_num
77             END IF
78          AFTER INSERT, DELETE
79             CALL renum_items()
80             CALL order_total()
81          AFTER ROW
82             CALL order_total()
83       END INPUT
84
85       IF int_flag THEN
86          LET int_flag = FALSE
87          CLEAR FORM
88          ERROR "Order input aborted" ATTRIBUTE (RED, REVERSE)
89          RETURN
90       END IF
91
92       CALL insert_order()
93    END FUNCTION
94
95    FUNCTION insert_order()
96       WHENEVER ERROR CONTINUE
97       BEGIN WORK
98       INSERT INTO orders (order_num, order_date, customer_num,
99          ship_instruct, po_num)
100         VALUES (0, p_orders.order_date, p_customer.customer_num,
101            p_orders.ship_instruct, p_orders.po_num)
102         IF status < 0 THEN
103            ROLLBACK WORK
104            ERROR "Unable to complete update of orders table"
105            ATTRIBUTE(RED, REVERSE, BLINK)
106            RETURN
107         END IF
108      LET p_orders.order_num = SQLCA.SQLERRD[2]
109      DISPLAY BY NAME p_orders.order_num
110         IF NOT insert_items() THEN
111            ROLLBACK WORK
112            ERROR "Unable to insert items" ATTRIBUTE(RED, REVERSE, BLINK)
113            RETURN
114         END IF
115      COMMIT WORK
116      WHENEVER ERROR STOP
117      CALL mess("Order added", 23)
```

```
118     CLEAR FORM
119  END FUNCTION
120
121
122  FUNCTION order_total()
123     DEFINE order_total MONEY(8),
124        i INTEGER
125
126     LET order_total = 0.00
127     FOR i =1  TO arr_count()
128        IF p_items[i].total_price IS NOT NULL THEN
129           LET order_total = order_total + p_items[i].total_price
130        END IF
131     END FOR
132     LET order_total = 1.1 * order_total
133     DISPLAY order_total TO t_price ATTRIBUTE (GREEN)
134  END FUNCTION
135
136
137  FUNCTION item_total()
138     DEFINE pa_curr, sc_curr INTEGER
139
140     LET pa_curr = arr_curr()
141     LET sc_curr = scr_line()
142     LET p_items[pa_curr].total_price =
143        p_items[pa_curr].quantity * p_items[pa_curr].unit_price
144     DISPLAY p_items[pa_curr].total_price TO s_items[sc_curr].total_price
145  END FUNCTION
146
147
148  FUNCTION renum_items()
149     DEFINE pa_curr, pa_total, sc_curr, sc_total, k INTEGER
150
151     LET pa_curr = arr_curr()
152     LET pa_total = arr_count()
153     LET sc_curr = scr_line()
154     LET sc_total = 4
155     FOR k = pa_curr TO pa_total
156        LET p_items[k].item_num = k
157        IF sc_curr <= sc_total THEN
158           DISPLAY k TO s_items[sc_curr].item_num
159           LET sc_curr = sc_curr + 1
160        END IF
161     END FOR
162  END FUNCTION
163
164
165  FUNCTION insert_items()
166     DEFINE idx INTEGER
167
168     FOR idx=1  TO arr_count()
169        IF p_items[idx].quantity != 0 THEN
170           INSERT INTO items
171              VALUES (p_items[idx].item_num, p_orders.order_num,
172                 p_items[idx].stock_num, p_items[idx].manu_code,
173                 p_items[idx].quantity, p_items[idx].total_price)
174              IF status < 0 THEN
175                 RETURN (FALSE)
176              END IF
177        END IF
178     END FOR
```

```
179     RETURN (TRUE)
180  END FUNCTION
181
182
183  FUNCTION get_stock()
184     DEFINE idx integer
185
186     OPEN WINDOW stock_w AT 7, 3
187        WITH FORM "stock_sel"
188        ATTRIBUTE(BORDER, YELLOW)
189     CALL set_count(stock_cnt)
190     DISPLAY " Use cursor using F3, F4, and arrow keys; press ESC ",
191        "to select a stock item" AT 1,1
192     DISPLAY ARRAY p_stock TO s_stock.*
193     LET idx = arr_curr()
194     CLOSE WINDOW stock_w
195     RETURN p_stock[idx].stock_num, p_stock[idx].manu_code,
196        p_stock[idx].description,  p_stock[idx].unit_price
197  END FUNCTION
198
199
200  FUNCTION get_item()
201    DEFINE pa_curr, sc_curr INTEGER
202
203     LET pa_curr = arr_curr()
204     LET sc_curr = scr_line()
205     SELECT description, unit_price
206        INTO p_items[pa_curr].description,
207           p_items[pa_curr].unit_price
208        FROM stock
209        WHERE stock.stock_num = p_items[pa_curr].stock_num
210           AND stock.manu_code = p_items[pa_curr].manu_code
211        IF status THEN
212           LET p_items[pa_curr].description = NULL
213           LET p_items[pa_curr].unit_price = NULL
214        END IF
215        DISPLAY p_items[pa_curr].description, p_items[pa_curr].unit_price
216           TO s_items[sc_curr].description, s_items[sc_curr].unit_price
217        IF p_items[pa_curr].quantity IS NOT NULL THEN
218           CALL item_total()
219        END IF
220  END FUNCTION
221
222
223  FUNCTION find_order()
224     ERROR "Function not yet implemented"
225     SLEEP 3
226     RETURN
227  END FUNCTION
```

The order module references the **globals** file with the GLOBALS statement and defines a module variable, **query_stat**. It consists of nine functions that collect, manipulate, and insert the order data:

- The **add_order** function calls the **query_customer** function and determines whether or not the user has selected a customer. If there is an active customer, it assigns values to three of the four variables of the **p_orders** program record, from the data values entered by the user into the **order_date**, **po_num**, and **ship_instruct** fields. It then allows the user to place up to 10 items in the order. Four items can be displayed on the screen at one time. The user can press CTRL-B while the cursor is in either the **stock_num** or the **manu_code** field to see a list of available stock items and select an item directly from the list presented.
- The **insert_order** function is called from the **add_order** function to perform a transaction that adds an order to the database.
- The **order_total** function is called from the **add_order** function to add the values in the **total_price** column of the **p_items** program array and to display the sum on the screen form.
- The **item_total** function is called from the **add_order** function or the **get_item** function to compute and display a total price for the current item (the current row of the **p_items** program array).
- The **renum_items** function is called from the **add_order** function to renumber the items in the program array and the screen array if the user adds or deletes a row in the screen array.
- The **insert_items** function is called from the **insert_order** function to add rows to the **items** table. Each row contains an order number (**p_orders.order_num**) and values from the **p_items** program array. The **insert_items** function returns a value to the calling function to indicate whether the rows were successfully added.
- The **get_stock** function displays the stock list retrieved by the **fetch_stock** function within a window on the screen form. When the user presses ESC to select an item from the list, the function returns the item values in the current row of the **p_stock** program array.

- If the user enters a stock number and manufacturer code, the **get_item** function is called from the **add_order** function to look up and display a description and unit price for the corresponding item. If the user has also entered a quantity, the **get_item** function calls **item_total** to compute and display a total item price.
- The **find_order** function displays the message that the function is not yet implemented.

Defining and Compiling the Program

This section shows you how to define and compile the **cust_order** program from the Programmer's Environment.

You should first compile the two screen forms required by the program, **orderform.per** and **stock_sel.per**.

To compile the screen forms from the Programmer's Environment

1. Enter `r4gl` at the system prompt to access the **INFORMIX-4GL** menu.
2. Choose the **Form** option.
3. Choose the **Compile** option from the **FORM** menu.
4. Select the **orderform** form.

   The message is displayed while the form compiles:

   ```
   Form compilation in progress...please wait.
   ```
5. Repeat the compilation process for the **stock_sel** form.
6. Choose the **Exit** option when finished to return to the **INFORMIX-4GL** menu.

To define the cust_order program from the Programmer's Environment

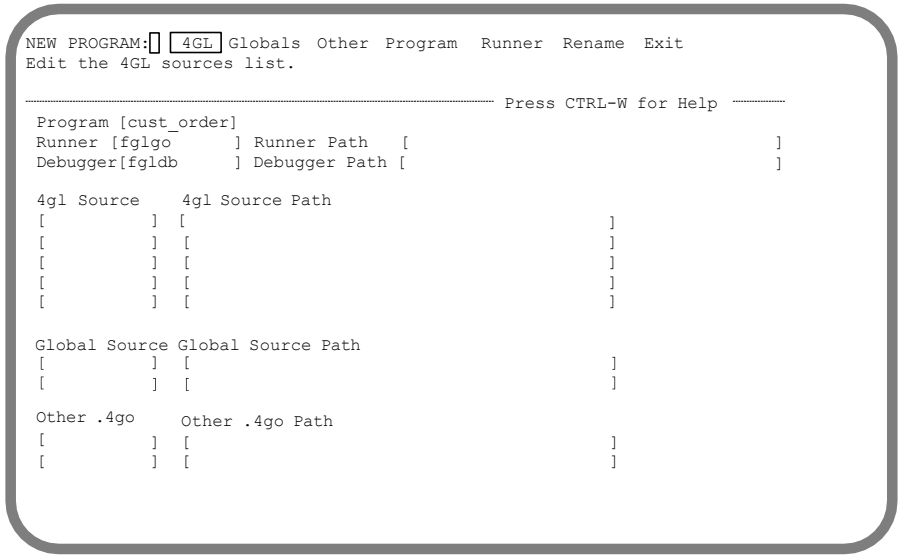
1. Choose the **Program** option from the **INFORMIX-4GL** menu.
2. Choose the **New** option from the **PROGRAM** menu.

3. Enter the following name when prompted:

```
cust_order
```

If you are using the **Program** menu option for the first time, you will be prompted to create the program database before you can proceed. Answer `y` to this question if it appears. See the *INFORMIX-4GL Reference* for information on the program database. After a few seconds, the NEW PROGRAM screen appears, as shown in Figure 5-1.

Figure 5-1
The NEW PROGRAM Screen



The name of the program appears as **cust_order**. The program consists of one **Globals** module and two 4GL source modules.

To enter the 4GL source modules

1. Choose the **4GL** menu option.
2. Enter `main` as the first 4GL source module.
3. Enter `order` as the second 4GL source module and press ESC.

To enter the Globals module

1. Choose the **Globals** option.
2. Enter `globals` as the global source module and press ESC.

   The NEW PROGRAM screen appears, as shown in Figure 5-2.

Figure 5-2
Defining the cust_order Program

```
NEW PROGRAM:[ 4GL  Globals  Other  Program Runner  Rename  Exit
Edit the 4GL sources list.

                                                Press CTRL-W for Help
 Program [cust_order]
 Runner [fglgo      ] Runner Path   [                                        ]
 Debugger[fgldb     ] Debugger Path [                                        ]

 4gl Source     4gl Source Path
 [main       ]  [                                          ]
 [order      ]  [                                          ]
 [           ]  [                                          ]
 [           ]  [                                          ]
 [           ]  [                                          ]

 Global Source Global Source Path
 [globals    ]  [                                          ]
 [           ]  [                                          ]

 Other .4go     Other .4go Path
 [           ]  [                                          ]
 [           ]  [                                          ]
```

Compiling the Program

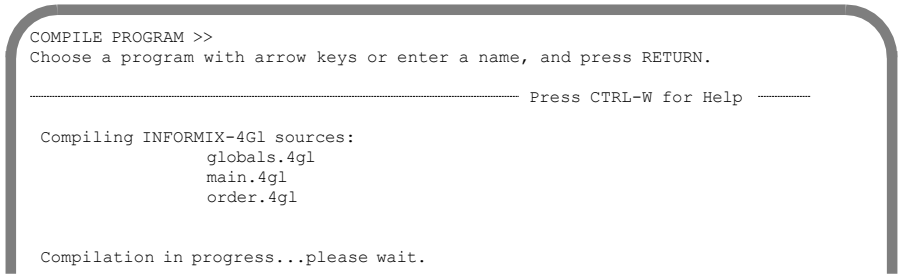
Now that you have defined the programs, the next step is to compile it.

To compile the program

1. Choose the **Exit** option to return to the **PROGRAM** menu.
2. Choose the **Compile** option to compile the three modules into the **cust_order** executable program.

   The name of each module is displayed on the screen individually as the module compiles. Figure 5-3 illustrates the appearance of the screen when the final module has begun compilation.

Figure 5-3
Compiling the cust_order Program



When the process is complete, you should see the following message:

```
Program successfully compiled.
```

For information on compiling a multi-module program from the command line and for more information on the process of compilation, see the *INFORMIX-4GL Reference*.

Working with Multi-Module Programs

Once the program is compiled, you are ready to access the Debugger.

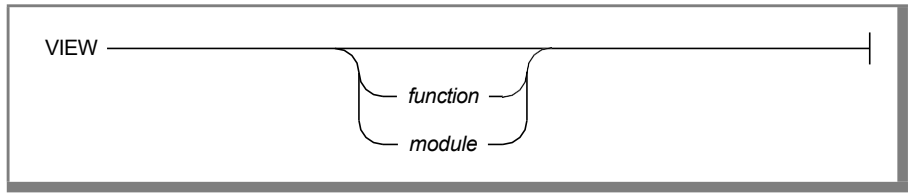
To initiate the debugging session

1. Choose the **Debug** option from the **PROGRAM** menu.
2. Select the **cust_order** program.

After a few seconds, the Source and Command windows appear on the terminal screen as shown in Figure 5-4.

Figure 5-4
Accessing the Debugger

Viewing a Module in the Source Window



When you first access a multi-module program using the Debugger, the Source window displays the module that contains the MAIN section. On a standard terminal, the first nine lines of the MAIN section appear in the Source window, followed by the name of the module and current function.

You can display a different module in the Source window by using the VIEW command followed by the module name. Do not use the **.4gl** extension when entering a module name.

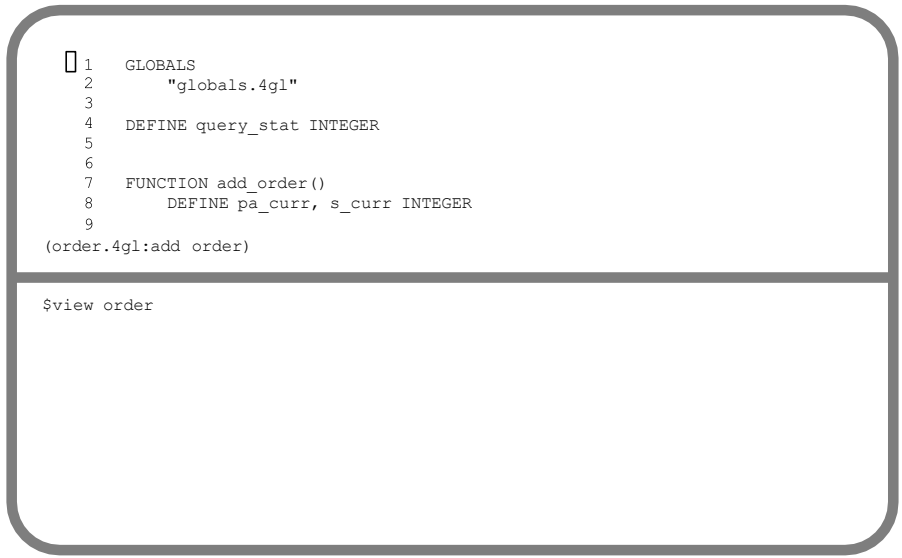
To display a different module

1. Enter the following command to display the **order** module in the Source window:

```
view order
```

   The Debugger displays the first nine lines of the module you have specified in the Source window, and the last line of this window changes to reflect the current module and function as shown in Figure 5-5.

Figure 5-5
Viewing a Different Module in the Source Window



2. Press the **Interrupt** key (CTRL-C) to return the cursor to the Command window.

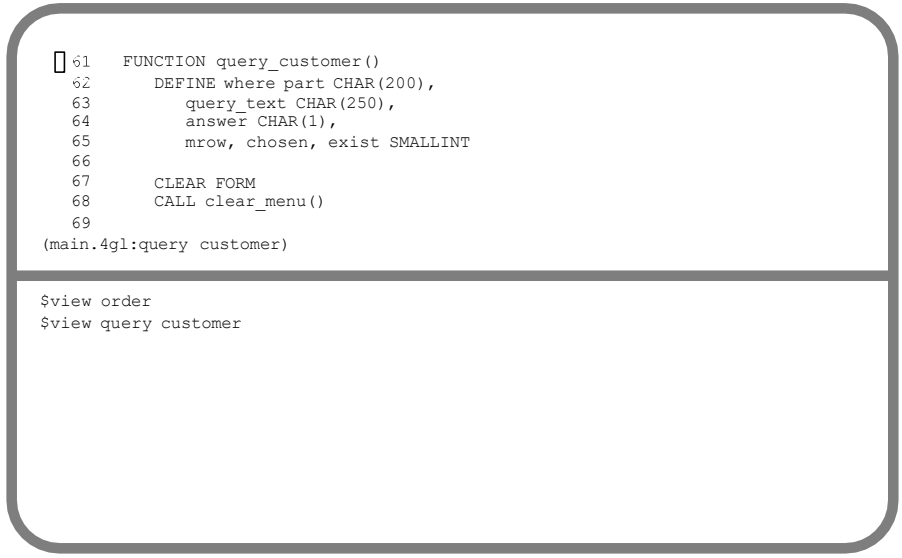
Viewing a Function in a Different Module

You do not need to know the module in which a specific function is located in order to display it. For example, to display the **query_customer** function in the Source window, enter the following command:

```
view  query_customer
```

This command restores **main** as the current module and makes **query_customer** the current function.

Figure 5-6
Viewing a Function in a Different Module



Searching the Current Module

You can only search for a pattern of characters within the current module. See Chapter 2 if you would like to review the search characters or any of the other keys available for manipulating the Debugger windows.

To search the current module

1. Use the appropriate keys to scroll the **main** module through the Source window and to search for patterns of characters within this module.
2. Press the **Interrupt** key (CTRL-C) when finished to return the cursor to the Command window.

Setting Tracepoints or Breakpoints at a Line Number

To set a tracepoint or breakpoint at a line number in a multi-module program, you must do one of the following things:

- Use the VIEW command to make the module in which the line number occurs the current module.
- Qualify the line number with the module name.

In the **cust_order** program, if **main** is the module currently displayed in the Source window, you must enter the following command to set a breakpoint at line 192 of the **order** module:

```
break order.192
```

If **order** is the module currently displayed in the Source window, you can enter the following command:

```
break 192
```

This breakpoint causes the Debugger to suspend execution immediately prior to executing the DISPLAY ARRAY statement at line 192 of the **order** module.

Module Variables

A module variable is a variable defined outside the GLOBALS statement, and before any of the functions in a module. The scope of a module variable is the module in which it occurs. In the **order** module, the **query_stat** module variable is defined as follows:

```
1   GLOBALS
2        "globals.4gl"
3
4      DEFINE  query_stat  INTEGER
```

Setting Tracepoints or Breakpoints on Module Variables

To set a tracepoint or breakpoint on a module variable, you must do one of the following things:

- Use the VIEW command to make the module in which the variable is defined the current module.
- Qualify the variable name with the keyword MODULE followed by the module name.

In the **cust_order** program, if **order** is the module currently displayed in the Source window, you can enter the following command to set a tracepoint on **query_stat**:

```
trace  query_stat
```

If **main** is the module currently displayed in the Source window, you must enter the following command to set the tracepoint:

```
trace   module.order.query_stat
```

You must qualify a module variable if there is a local variable with the same name defined in the current function.

See "Scope of Reference" on page 9-16 in for more information on how to qualify variables in the Debugger.

Module Variables and the DUMP Command

The DUMP command displays the values of module variables if you use either the GLOBALS or the ALL option. Module variables are displayed following global variables and before local variables.

For example, in the **cust_order** program, the module variable **query_stat** would appear in the output of the DUMP ALL command as follows:

```
DUMPING  GLOBAL  VARIABLES
         ...

         ...
DUMPING  GLOBAL  VARIABLES  OF  MODULE  [order]

   query_stat  =  1

DUMPING  LOCAL  VARIABLES  OF  FUNCTION  [add_order]
         ...

         ...
```

This chapter summarizes the facts that you should be aware of when you work with multi-module programs. When you work within the Programmer's Environment, you must define a multi-module program by using the **New Program** menu option before you can compile and run it. You can use the VIEW command with the *module* option to display a different module in the Source window. The Debugger defaults to the module currently displayed in the Source window when you set tracepoints or breakpoints at a line number. When setting tracepoints or breakpoints on a module variable, you must either make the module in which the variable is defined the current module or qualify the variable with the keyword MODULE followed by the module name.

Chapter

6

Tracing Logic of the cust_order Program

In This Chapter

This chapter uses the **cust_order** program created in Chapter 5, "A Multi-Module Program: cust_order," to introduce additional commands and capabilities of the Debugger, and to illustrate further ways in which you can interact with an executing INFORMIX-4GL program. The following topics are covered in this chapter:

- How to set tracepoints and breakpoints without enabling them
- How to use the ENABLE command to activate a tracepoint or breakpoint
- Options for the STEP command when the next statement is a function call
- How to use the AUTOTOGGLE terminal display parameter to alter the conditions under which the Debugger displays the Application screen
- How to use the CALL command to execute a function interactively
- How to redirect the output of the TRACE and PRINT commands to a file

Two intentional bugs produce fatal errors in the **cust_order** program. The examples in this chapter are designed to help you familiarize yourself with the program by using aspects of it that are working correctly. Chapter 7, "Analyzing Runtime Errors in the cust_order Program," shows you how to produce the fatal errors, and illustrates the operation of the Debugger when runtime errors occur. If you produce a fatal error in the course of working with the program in this chapter, you should turn to Chapter 7 for information on how to correct it and restart the Debugger.

The examples in this chapter emphasize the functions of the **order** module that manipulate the values entered onto the screen form and insert them into the database. Chapter 7 emphasizes the functions of the **main** module.

Overview of the Debugging Session

You are already familiar with two principal debugging tools, the setting of tracepoints and breakpoints. Specifically, you have learned how to perform the following operations:

- Set tracepoints and breakpoints with the TRACE and BREAK commands, respectively.
- Monitor the operation of a program with tracepoints and breakpoints set.
- Make a tracepoint or breakpoint inactive with the DISABLE command.
- Remove tracepoints and breakpoints with the NOTRACE and NOBREAK commands, respectively.

It is also possible to define a tracepoint or breakpoint without enabling it. This allows you, for example, to define multiple tracepoints and breakpoints at the beginning of a session or in a **.4db** file and to enable, or activate, them as needed.

In setting up the debugging session for this chapter, you will carry out the following activities:

- Define two enabled tracepoints.
- Define two breakpoints without enabling them.

Each tracepoint or breakpoint is designed to familiarize you with specific program segments or activities and to illustrate a particular Debugger command or strategy.

The **cust_order** program is more complex than the **customer** program introduced in Chapter 2, "Getting Started with the Debugger," and this chapter cannot examine its operation with the same degree of detail. Consult Appendix C, "Sample Programs," if you would like more information on the program segments described in the course of this chapter.

Setting Tracepoints for the Current Session

The tracepoints that you set in this session monitor the functions that insert the order and item information into the database. These tracepoints use the PRINT command to display the values inserted by the traced functions into the **orders** and **items** tables.

The tracepoints illustrate how you can write the output of the TRACE and PRINT commands to a file. Because the combined number of lines generated by these tracepoints exceeds the number of lines that the Command window retains in its buffer (50), writing to a file is the only way to preserve a complete record of their output after the tracepoints have executed.

Setting the First Tracepoint

The first tracepoint allows you to monitor the execution of the **insert_order** function and to print out the values inserted by this function into the **orders** table. The **insert_order** function executes after the user has pressed ESC to terminate order entry.

To set this tracepoint, enter the following command:

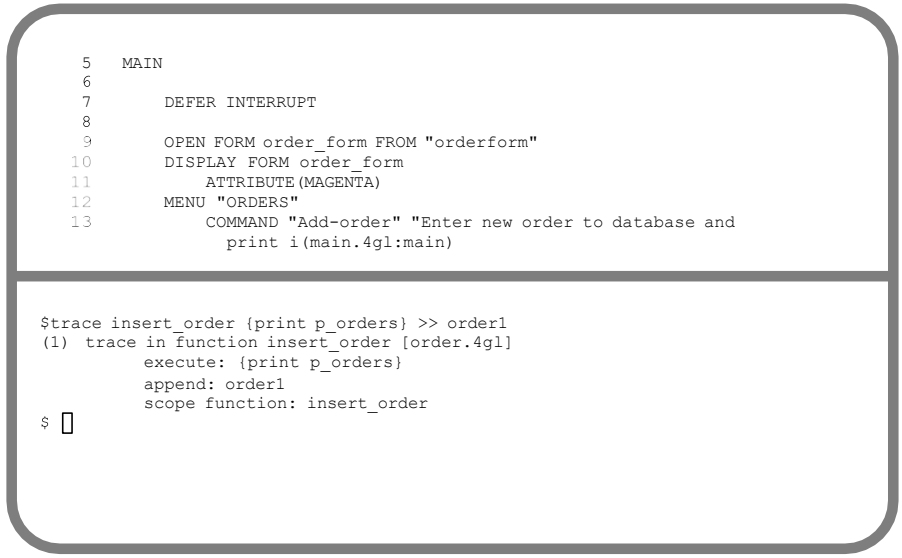
```
trace insert_order {print p_orders} >> order1
```

This command tells the Debugger to perform the following operations:

- Output the line numbers at which the **insert_order** function begins and ends execution.
- Print the values of the members of the **p_orders** program record when the **insert_order** function is called.
- Write these values to a file named **order1**.

Figure 6-1 illustrates your entry of the first tracepoint and the Debugger response.

Figure 6-1
Setting the First Tracepoint



Setting the Second Tracepoint

The second tracepoint allows you to monitor the execution of the **insert_items** function and to print out the values inserted by this function into the **items** table. The **insert_items** function is called by **insert_order** as follows:

```
95  FUNCTION insert_order()
          ...
110   IF NOT insert_items() THEN
111      ROLLBACK WORK
112      ERROR "Unable to insert items" ATTRIBUTE(RED, REVERSE,BLINK)
113      RETURN
114   END IF
          ...
119   END FUNCTION
```

The **insert_items** function returns a value of `1`, or TRUE, to **insert_order** if the values are successfully inserted, and a value of `0`, or FALSE, otherwise.

To set this tracepoint, enter the following command:

```
trace insert_items {print p_items} >> order1
```

This command tells the Debugger to perform the following operations:

- Output the line numbers at which the **insert_items** function begins and ends execution.
- Output the value returned by **insert_items** to the calling function.
- Print the values of the members of each element of the **p_items** program array when the **insert_items** function is called.
- Write these values to a file named **order1**.

Figure 6-2 illustrates your setting of the second tracepoint, and the Debugger response.

Figure 6-2
Setting the Second Tracepoint

```
 5   MAIN
 6
 7       DEFER INTERRUPT
 8
 9       OPEN FORM order_form FROM "orderform"
10       DISPLAY FORM order_form
11           ATTRIBUTE(MAGENTA)
12       MENU "ORDERS"
13           COMMAND "Add-order" "Enter new order to database and
               print i(main.4gl:main)
```

```
(1)  trace in function insert_order [order.4gl]
          execute: {print p_orders}
          append: order1
          scope function: insert_order
$trace insert_items {print p_items} >> order1
(2)  trace in function insert_items [order.4gl]
          execute: {print p_items}
          append: order1
          scope function: insert_items
$
```

Setting Breakpoints for the Current Session

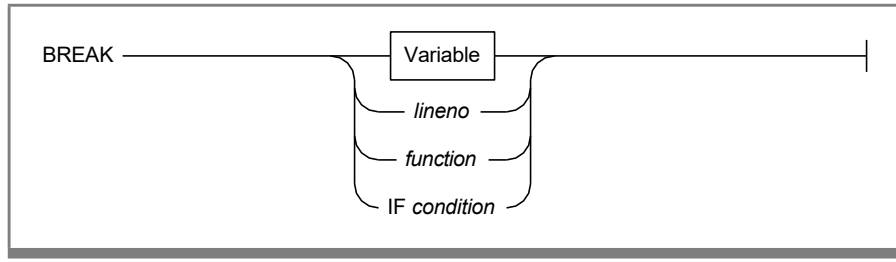
The first breakpoint that you set for this session is designed to help you understand the program segments that perform the important operations of calculating the item and order totals for a particular customer order.

The second breakpoint is designed to let you observe the feedback the program provides if invalid data values are entered and to let you compare different strategies for changing the values.

These breakpoints are disabled, or inactive, when you set them. They are designed to give you additional practice in working with breakpoints and to show how breakpoints can be enabled and disabled as needed in the course of a debugging session.

Setting Tracepoints and Breakpoints Without Enabling Them

You use an asterisk (*) following the command name to define a tracepoint or breakpoint without enabling it. The general format for setting a breakpoint without enabling it is as follows.



Setting the First Breakpoint

The first breakpoint is designed to familiarize you with the program segments that calculate and display the item and order totals. It causes program execution to be suspended when the **item_total** function is called.

To set a breakpoint when a function is called, follow the BREAK keyword with the function name. Do not include parentheses after the function name.

Enter the following command to set the first breakpoint:

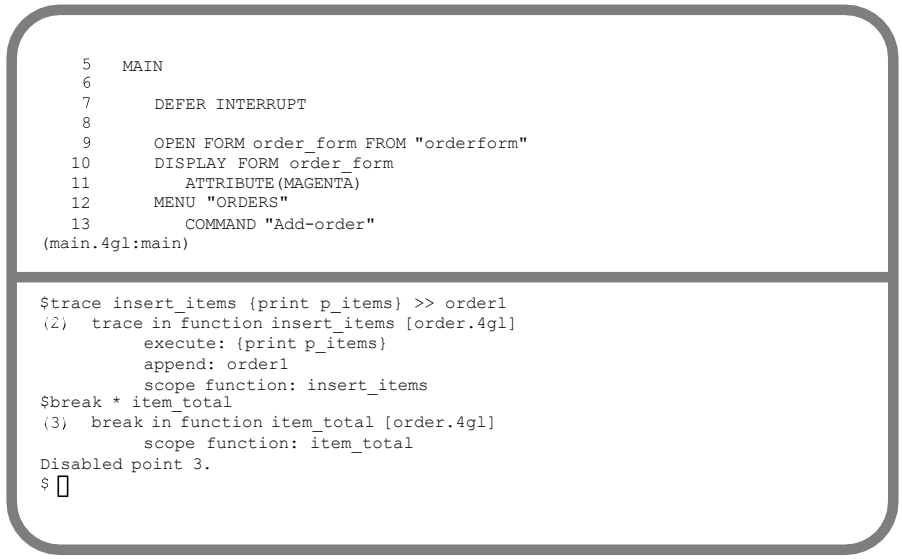
```
break * item_total
```

This command tells the Debugger to perform the following tasks:

- Suspend program execution when the **item_total** function is called.
- Make the breakpoint inactive until enabled by you.

Figure 6-3 illustrates your entry of this command and the Debugger response.

Figure 6-3
Setting the First Breakpoint



The asterisk (*) does not appear when the Debugger echoes the breakpoint. However, the Debugger displays the information that the breakpoint is disabled. The **item_total** function is called by **add_order** in an AFTER FIELD clause as follows:

```
66  AFTER  FIELD  quantity
67       MESSAGE  ""
68       LET  pa_curr  =  arr_curr()
69       IF  p_items[pa_curr].unit_price  IS  NOT  NULL
70          AND  p_items[pa_curr].quantity  IS  NOT  NULL
71       THEN
72          CALL  item_total()
```

This function multiplies the number entered in the **Quantity** field by the price of the item.

Setting the Second Breakpoint

A second breakpoint suspends program execution if the value of the 4GL global variable **status** is set to `100`, or NOTFOUND. For example, the value of **status** is set to NOTFOUND when the program attempts to verify invalid values entered into the **Stock No.** and **Code** fields on the screen form. Setting this breakpoint allows you to observe the feedback provided by the program when invalid values are entered. You will then compare two strategies for correcting invalid data and resetting the value of **status** to `0`. These strategies are as follows:

- Correct the values using the regular program mechanisms
- Correct the values using Debugger commands

This test for the value of **status** occurs at line 211 in the **get_item** function of the **order** module as follows:

```
205   SELECT  description,  unit_price
206      INTO   p_items[pa_curr].description,
207         p_items[pa_curr].unit_price
208      FROM  stock
209      WHERE  stock.stock_num  =  p_items[pa_curr].stock_num
210         AND  stock.manu_code  =  p_items[pa_curr].manu_code
211      IF  status  THEN
212         LET  p_items[pa_curr].description  =  NULL
213         LET  p_items[pa_curr].unit_price  =  NULL
214      END  IF
```

The **query_customer** function in the **main** module also performs a number of tests to determine if **status** is equal to `100`. However, this breakpoint is not enabled, or activated, in the debugging session until after **query_customer** executes.

To set the second breakpoint, enter the following command:

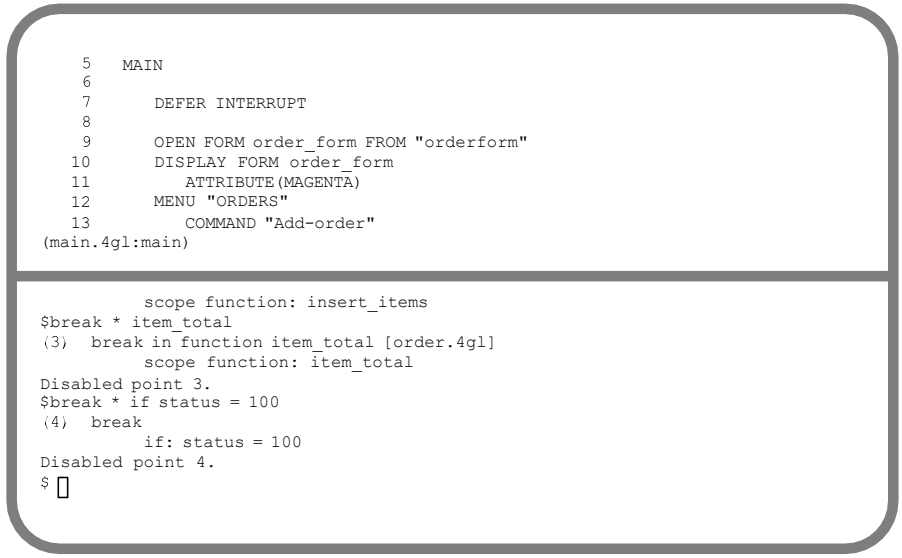
```
break * if status = 100
```

This command tells the Debugger to perform the following operations:

- Suspend program execution if the value of **status** is set to `100`.
- Make this breakpoint inactive until enabled by you.

Figure 6-4 illustrates your entry of this command and the Debugger's response.

Figure 6-4
Setting the Second Breakpoint



When you begin operation of the Debugger for this session, all of the terminal display parameters have their default values. You can use the LIST DISPLAY command if you would like to review these parameters and their values.

Tracing Program Logic: Example #1

The debugging steps in this section are designed to help you understand the program segments that calculate and display the item and order totals. In working with the program in this section, you will carry out the following activities:

- Choose the **Add-Order** option from the **cust_order** program menu.
- Enter search criteria and select a customer for whom to place an order.
- Enter sample order information for this customer.
- Add the first item to the order.

You will perform the following Debugger activities:

- Enable the first breakpoint you have set:

```
break item_total
```

- Enter the RUN command to start operation of the Debugger.
- Use the TURN command to switch the AUTOTOGGLE terminal display parameter OFF.
- Use the STEP command to step through individual 4GL statements.
- Use the STEP command to step over a function, treating all the statements in the function as a single step.
- Use the STEP command with the INTO option to step into a function, treating each executable statement in the function as a single step.

The ENABLE Command

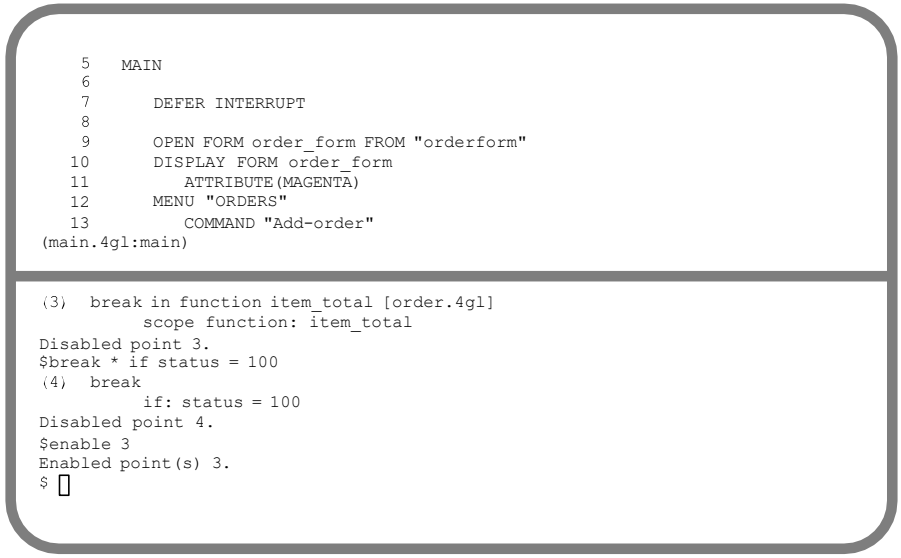
You can use the ENABLE command to make a disabled tracepoint or breakpoint active. When a tracepoint or breakpoint is enabled, the action it specifies continues until the point is disabled or removed.

You can refer to tracepoints or breakpoints by their reference numbers when enabling them. For example, enter the following command to enable the breakpoint with the reference number (3), **break item_total**:

```
enable 3
```

Figure 6-5 illustrates your enabling of this breakpoint and the Debugger response.

Figure 6-5
Enabling the First Breakpoint



With this breakpoint enabled, the Debugger causes program execution to be suspended as soon as the **item_total** function is entered.

Starting the Session

Starting the session involves starting the Debugger and selecting the customer for whom you want to place an order.

To start the session

1. Enter the RUN command or press F5 to begin operation of the Debugger.

   The Debugger displays the Application screen with the **cust_order** program menu and the **orderform** screen form, as shown in Figure 6-6.

Figure 6-6
The Application Screen with the cust_order Program

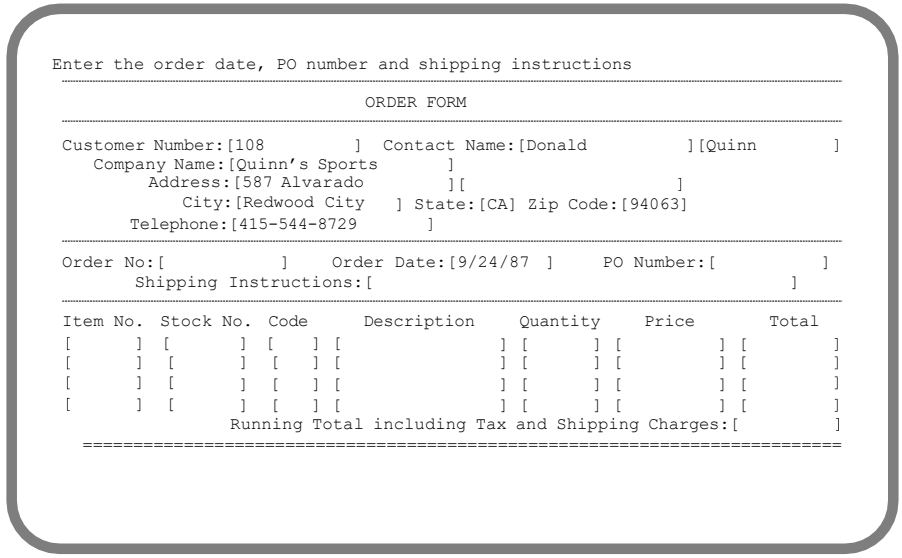
```
ORDERS: [ Add Order ] Find Order    Exit
Enter new order to database
-------------------------------------------------------------------------------

                              ORDER FORM

-------------------------------------------------------------------------------
  Customer Number:[           ]  Contact Name:[                ][             ]
     Company Name:[                    ]
          Address:[                    ][                    ]
             City:[                ] State:[  ] Zip Code:[      ]
        Telephone:[                   ]
-------------------------------------------------------------------------------
 Order No:[            ]    Order Date:[           ]    PO Number:[           ]
        Shipping Instructions:[                                          ]
-------------------------------------------------------------------------------
 Item No.  Stock No.  Code      Description     Quantity     Price        Total
 [       ] [       ] [    ] [                ] [       ] [          ] [         ]
 [       ] [       ] [    ] [                ] [       ] [          ] [         ]
 [       ] [       ] [    ] [                ] [       ] [          ] [         ]
 [       ] [       ] [    ] [                ] [       ] [          ] [         ]
                  Running Total including Tax and Shipping Charges:[          ]
  ============================================================================
```

2. Choose the **Add-Order** menu option.
3. Enter `108` in the **Customer Number** field to retrieve Donald Quinn and press ESC.
4. Choose the **Select** option from the **BROWSE** Menu to select this customer.

Warning: *Entering invalid customer search criteria or choosing the **Quit** option from the **BROWSE** menu produces a fatal error that aborts operation of the **cust_order** program. If you produce this error, you should reenter the RUN command and take care to retrieve and select a valid customer entry. You can turn to Chapter 7 for a discussion of this error and instructions for correcting it. Appendix A, "The Demonstration Database and Application," in "INFORMIX-4GL by Example" lists the valid customer entries for the **stores7** demonstration database.*

When Donald Quinn is selected, the Application screen appears as shown in Figure 6-7.

Figure 6-7
Selecting a Customer



Reaching the First Breakpoint

When a customer has been successfully retrieved and selected, the **add_order** function executes the statement at line 17:

```
17   INPUT  BY  NAME  p_orders.order_date,  p_orders.po_num,
18      p_orders.ship_instruct
```

The function waits for you to enter values into the **Order Date**, **PO Number**, and **Shipping Instructions** fields.

To enter order and item information

1. Enter some sample values into these fields and press RETURN after each entry.

   The order information you entered appears on the screen, as shown in Figure 6-8.

Figure 6-8
Entering the Order Information

```
Enter a stock number or press CTRL-B to scan stock list.
Press ESC to write order.
------------------------------------------------------------------------------

                              ORDER FORM

------------------------------------------------------------------------------
Customer Number: 108            Contact Name: Donald            Quinn
   Company Name: Quinn's Sports
        Address: 587 Alvarado
           City: Redwood City      State: CA  Zip Code: 94063
      Telephone: 415-544-8729
------------------------------------------------------------------------------
Order No:[           ]    Order Date: 09/24/1987     PO Number: JR1147
       Shipping Instructions: overnight delivery
------------------------------------------------------------------------------
Item No.  Stock No.  Code      Description     Quantity     Price        Total
[       ] [      ] [    ] [                ] [       ] [          ] [         ]
[       ] [      ] [    ] [                ] [       ] [          ] [         ]
[       ] [      ] [    ] [                ] [       ] [          ] [         ]
[       ] [      ] [    ] [                ] [       ] [          ] [         ]
                 Running Total including Tax and Shipping Charges:[          ]
  ============================================================================
```

   The **add_order** function now executes the statement at line 25:

```
25   INPUT ARRAY p_items FROM s_items.*
```

   The cursor is positioned at the **Stock No.** field of the first row of the screen array.

2. Enter the values that appear in Figure 6-9 into the **Stock No.** and **Code** field, and press RETURN after each entry.

Figure 6-9
Entering Item Information

```
Enter a stock number or press CTRL-B to scan stock list.
Press ESC to write order.

                                ORDER FORM

Customer Number: 108            Contact Name: Donald            Quinn
   Company Name: Quinn's Sports
        Address: 587 Alvarado
           City: Redwood City      State: CA  Zip Code: 94063
      Telephone: 415-544-8729

Order No:[           ]    Order Date: 09/24/1987     PO Number: JR1147
       Shipping Instructions: overnight delivery

Item No.  Stock No.  Code      Description     Quantity     Price        Total
[       ] [      5] [NRG] [                ] [       ] [          ] [          ]
[       ] [       ] [   ] [                ] [       ] [          ] [          ]
[       ] [       ] [   ] [                ] [       ] [          ] [          ]
[       ] [       ] [   ] [                ] [       ] [          ] [          ]
                  Running Total including Tax and Shipping Charges:[          ]
  ==============================================================================
```

Warning: *The screen displays the message* `Enter a stock number or press CTRL-B to scan stock list`. *Pressing* **CTRL-B** *to produce this stock list causes a fatal error. If you produce this error, you should reenter the* RUN *command, and enter values for the* ***Stock No.*** *and* ***Code*** *fields directly onto the screen form. You can turn to Chapter 7 for a discussion of this error and instructions for correcting it. See Appendix A "The Demonstration Database and Application," in "INFORMIX-4GL by Example," for a list of valid stock items in the* ***stores7*** *demonstration database.*

When you enter a value into either the **Stock No.** or the **Code** field, **add_order** performs a test to determine if you have entered a value for both. Item entry does not proceed until both fields have a value because it is the combination of a stock number and manufacturer code that uniquely identifies an item in the **stock** table. See Appendix A *"The Demonstration Database and Application,"* in *INFORMIX-4GL by Example* for more information on the columns in the **stores7** database.

Because you have entered valid values in both fields, the program displays the description and price of the item on the form. These values are as shown in Figure 6-10.

Figure 6-10
Displaying Item Information

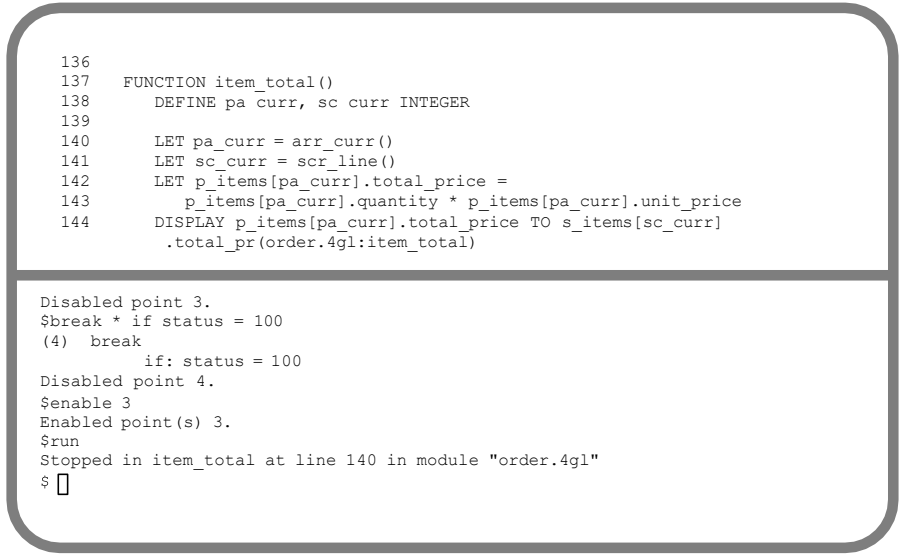
```
Enter the item quantity
Press ESC to write order
------------------------------------------------------------------------------

                              ORDER FORM
------------------------------------------------------------------------------

Customer Number: 108            Contact Name: Donald            Quinn
   Company Name: Quinn's Sports
        Address: 587 Alvarado
           City: Redwood City      State: CA  Zip Code: 94063
      Telephone: 415-544-8729
------------------------------------------------------------------------------

Order No:[           ]    Order Date: 09/24/1987     PO Number: JR1447
       Shipping Instructions: overnight delivery
------------------------------------------------------------------------------

Item No.  Stock No.  Code      Description     Quantity     Price        Total
[       ] [      5] [NRG] [tennis racquet ] [ ]      ] [    $28.00] [         ]
[       ] [       ] [    ] [               ] [       ] [          ] [         ]
[       ] [       ] [    ] [               ] [       ] [          ] [         ]
[       ] [       ] [    ] [               ] [       ] [          ] [         ]
                  Running Total including Tax and Shipping Charges:[         ]
  ============================================================================
```

The cursor is now positioned at the **Quantity** field, and the program is awaiting input. Therefore, enter `12` as the quantity, and press RETURN.

As soon as the cursor leaves the **Quantity** field, **add_order** executes the block of code following the AFTER FIELD **quantity** clause and calls the **item_total** function.

Because you have set a breakpoint when this function is called, program execution is suspended. Control returns to the Debugger, which redisplays the Source and Command windows as shown in Figure 6-11.

Figure 6-11
Reaching the First Breakpoint



When you set a breakpoint on a function, program execution stops as soon as the function is called. The Source window highlights the first executable statement in the function. The first executable statement in the **item_total** function is the statement at line 140:

```
140   LET pa_curr = arr_curr()
```

This is therefore the first statement to execute when operation resumes.

The Command window displays the function name, line number, and module name at which the breakpoint occurred. This is line 140 in the **item_total** function in the **order** module.

Resuming Operation Following the First Breakpoint

You have set a breakpoint at this function in order to suspend execution and examine in detail the program segments that calculate and display the item and order totals. Item totals are calculated by the current function, **item_total**. The listing of this function follows:

```
137  FUNCTION item_total()
138     DEFINE pa_curr, sc_curr INTEGER
139
140     LET pa_curr = arr_curr()
141     LET sc_curr = scr_line()
142     LET p_items[pa_curr].total_price =
143        p_items[pa_curr].quantity * p_items[pa_curr].unit_price
144     DISPLAY p_items[pa_curr].total_price TO s_items[sc_curr].total_price
145  END FUNCTION
```

In "Stepping Through a Function" on page 6-21, you will execute these statements line by line. Before you do this, however, it is desirable to alter the conditions under which the Debugger toggles the Application screen.

The AUTOTOGGLE Parameter

Under default conditions, the Debugger switches from the Debugger screen to the Application screen when either of the following conditions occurs:

- The program requires input.
- The program generates output.

If you are examining a series of 4GL statements, or performing diagnostic tests, it is often distracting to have the Application screen toggle every time the program produces output. You can use the AUTOTOGGLE terminal display parameter to control the conditions under which the Application screen appears.

When the AUTOTOGGLE parameter is set to OFF, the Application screen appears only when the program requires the input necessary to continue execution.

Enter the following command to prevent the Debugger from toggling to the Application screen every time the statements you are examining generate output:

```
turn off autotoggle
```

Stepping Through a Function

In Chapter 4, "Analyzing a Logical Error in the customer Program," you used the STEP command to execute a single 4GL statement. Stepping through statements one at a time or a few at a time is an excellent way to familiarize yourself with the flow of control in an unfamiliar program.

To step through a function

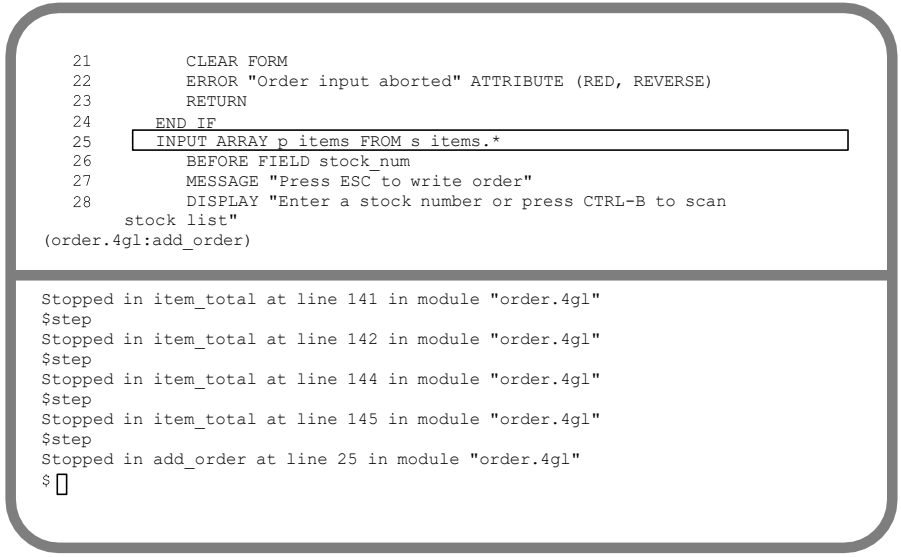
1. Enter the STEP command or press F2 five times in succession to move through the next five statements of the **item_total** function one at a time.

   These statements carry out the following activities:

   - Assign to the local variable **pa_curr** the value returned by the 4GL function **arr_curr**.
   - Assign to the local variable **sc_curr** the value returned by the 4GL function **scr_line**.
   - Multiply the values of **quantity** and **unit_price** in the current program array row and assign the result to **total_price**.
   - Display the value of **total_price** to the **Total** field on the screen form.
   - End the function.

   Each time you enter the STEP command, the Debugger highlights the next statement to be executed in the Source window. It displays in the Command window the line number of the statement you have just executed and the name of the function and module in which it occurs. Figure 6-12 illustrates the appearance of the Source and Command windows when you have performed this series of five steps.

Figure 6-12
Stepping Through the item_total Function



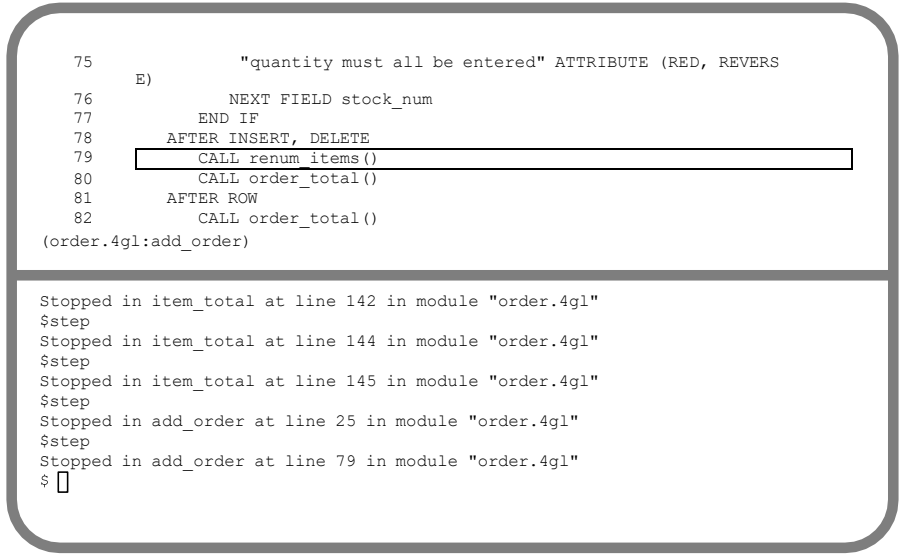
When you execute the final statement in the **item_total** function, you see that control returns to the calling function **add_order**. The Source window highlights the following statement:

```
25   INPUT ARRAY p_items FROM s_items.*
```

2. Enter the STEP command or press F2 to execute this statement.

The Debugger toggles briefly to the Application screen when it executes the INPUT ARRAY statement. It then redisplays the Source and Command windows with the contents, shown in Figure 6-13.

Figure 6-13
Stepping Through the INPUT ARRAY Statement



Stepping over a Function

At this stage, you have entered all the values for the first item on the order. Two functions are called following the AFTER INSERT clause at line 78. The Debugger highlights the first of the function calls in the Source window:

```
79   CALL  renum_items()
```

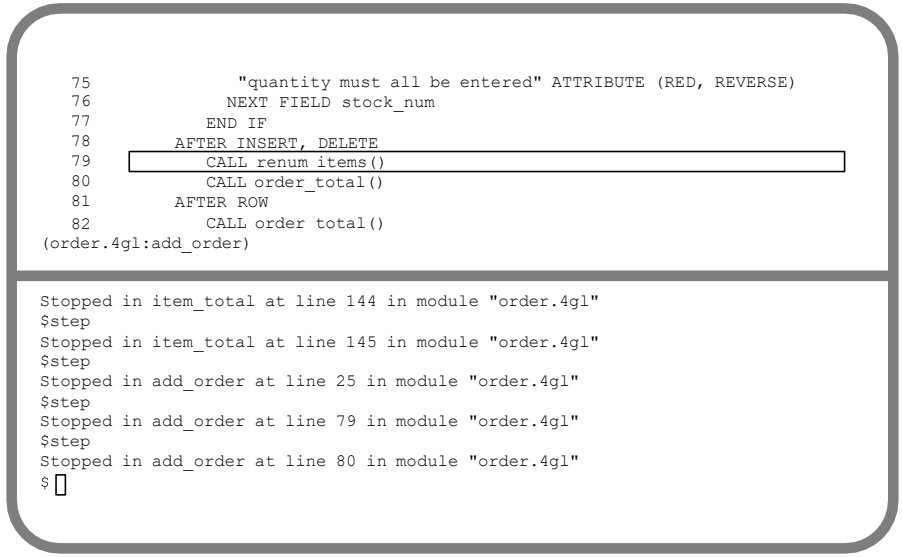
The **renum_items** function computes the item number associated with each item on the order, and displays this number in the **Item No.** field on the screen. Because the goal of this series of debugging steps is to observe the statements that calculate and display the item and order totals, you do not want to examine this function at this time.

When the statement you execute with the STEP command is a function call, all of the statements in the function are treated as a single step.

Enter the STEP command, or press F2, to tell the Debugger to execute all of the statements in the **renum_items** function as a unit.

The **renum_items** function executes, and control returns to **add_order**. Figure 6-14 illustrates your entry of the STEP command and the Debugger output.

Figure 6-14
Stepping over the renum_items Function



Stepping into a Function

The Source window now highlights the second of the two function calls that occur following the AFTER INSERT clause. This is the statement at line 80:

```
80   CALL order_total()
```

The **order_total** function calculates and displays the running total for the order. The listing of this function follows:

```
122  FUNCTION order_total()
123     DEFINE order_total MONEY(8),
124        i INTEGER
125
126     LET order_total = 0.00
127     FOR i = 1  TO ARR_COUNT()
128        IF p_items[i].total_price IS NOT NULL THEN
129        LET order_total = order_total + p_items[i].total_price
130        END IF
131     END FOR
132     LET order_total = 1.1 * order_total
133     DISPLAY order_total TO t_price ATTRIBUTE (GREEN)
134  END FUNCTION
```

Because you are monitoring the statements that compute and display the item and order totals, you want to examine the individual statements in this function.

When the next statement to execute is a function call, you can use the STEP command with the INTO option to tell the Debugger to treat the statements of the function as individual steps rather than as a unit. When you enter the STEP INTO command, the Debugger moves to the first executable statement in the function.

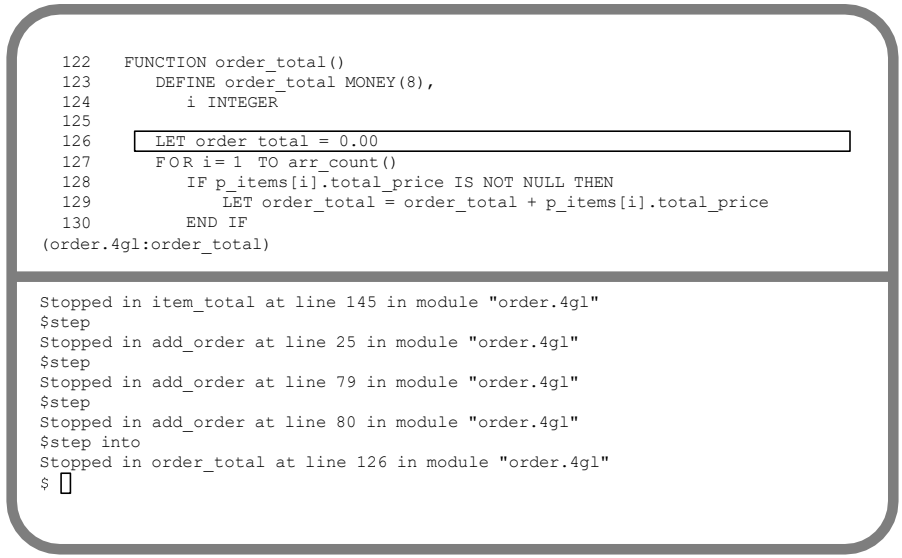
To use STEP INTO

1. Enter the following command or press F3 to tell the Debugger to treat the statements in the **order_total** function as individual steps:

```
step into
```

Figure 6-15 illustrates your entry of the STEP INTO command and the Debugger output.

Figure 6-15
Stepping INTO the order_total Function



The Source window highlights the first executable statement in the **order_total** function. This is the statement at line 126:

```
126   LET order_total = 0.00
```

2. Use the STEP command as you did with the **item_total** function to move line by line through **order_total**.

Observe that the statements in this function carry out the following activities:

- Initialize the MONEY variable **order_total** to 0.
- Use a FOR loop to assign a value to the **order_total** variable.
- Use the value returned by **arr_count** to set the upper boundary of the FOR loop to the total number of items in the **p_items** program array.

- Multiply the value assigned to **order_total** when the FOR loop terminates by 1.1 to add tax and shipping charges.
- Display the value of **order_total** to the field labeled **Running Total including Tax and Shipping Charges** on the screen form.
- End the function.

Because there is only one row entered at this time, the FOR loop executes only once, and the value returned by **arr_count** is `1`.

After you have used the STEP command to execute the final statement in the **order_total** function:

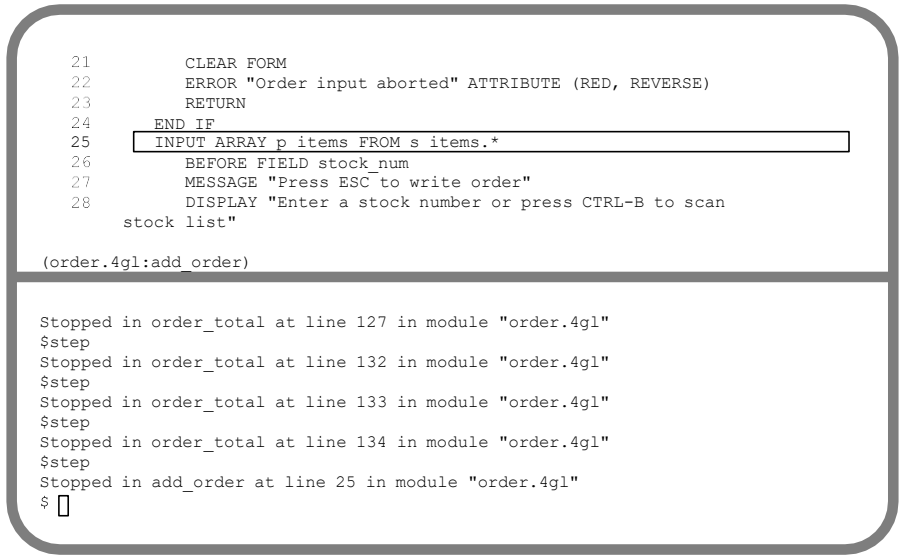
```
134   END FUNCTION
```

control returns to **add_order**. The first statement to execute when you resume operation in the next section is:

```
25   INPUT ARRAY p_items FROM s_items.*
```

The Debugger highlights this statement in the Source window as shown in Figure 6-16.

Figure 6-16
Stepping Through the order_total Function



This ends the first series of debugging steps. In the course of this section, you have entered the order information for a selected customer and entered the first row onto the order. You have enabled and reached the first breakpoint set in this chapter.

In the next section, you will modify the debugging environment and resume execution of the **cust_order** program with the INPUT ARRAY statement.

Tracing Program Logic: Example #2

The next series of debugging steps is designed to let you observe the feedback provided by the program when an invalid item is entered in a row, and to let you compare different strategies for producing a correct **Stock No.** and **Code** field combination.

In working with the program in this section, you will carry out the following activities:

- Enter invalid data, and observe the program response.
- Add two new rows to the customer order.
- Press ESC to terminate order entry.

You will perform the following Debugger activities:

- Modify the existing debugging environment.
- Resume operation with the CONTINUE command.
- Use the LET command to change the value of a variable.
- Use the CALL command to execute a function interactively.

Modifying the Debugging Environment

Before proceeding with the steps in this section, you should make certain changes in the debugging environment. Specifically, the changes you want to make are as follows:

- Disable the first breakpoint, **break item_total**. This breakpoint has a reference number of (3).
- Enable the second breakpoint, **break if status = 100**. This breakpoint has a reference number of (4).
- Return the value of AUTOTOGGLE to its default value of ON.

To modify the debugging environment

1. Enter the following command to disable the first breakpoint:

   ```
   disable 3
   ```

   Now the Debugger will not suspend program execution when the **item_total** function is called. This allows you to observe more clearly the action of the second breakpoint.

2. Enter the following command to enable the second breakpoint:

   ```
   enable 4
   ```

   Now the Debugger will suspend program execution if the value of **status** is set to `100`.

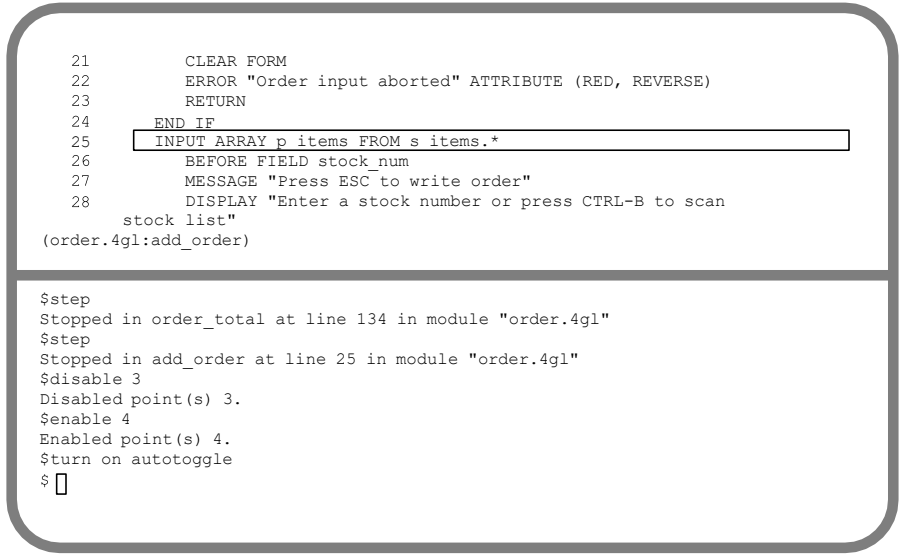
3. Enter the following command to change the value of AUTOTOGGLE to ON:

   ```
   turn on autotoggle
   ```

   Now the Debugger will display the Application screen both when the program requires input and when it generates output.

   Figure 6-17 illustrates your entry of these three commands and the Debugger response to each.

Figure 6-17
Modifying the Debugging Environment



Resuming Execution

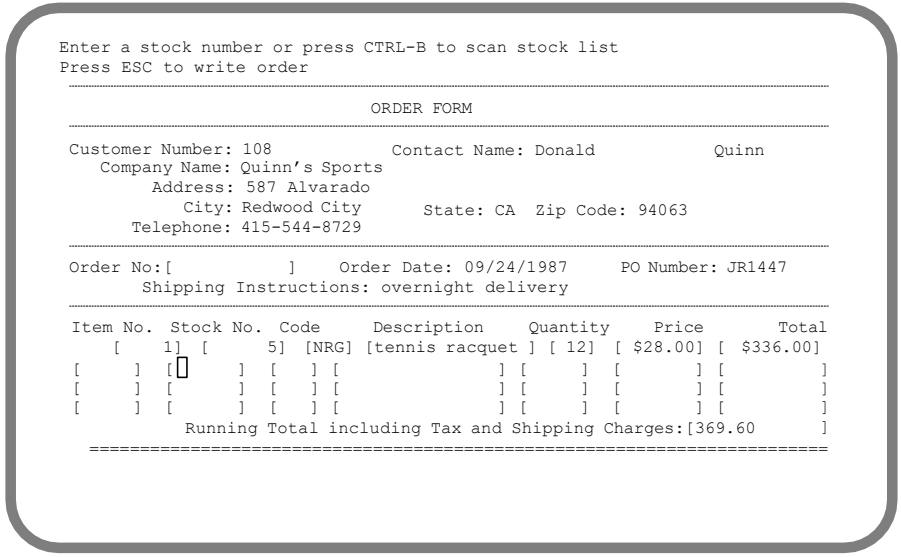
The Source window currently highlights the statement at line 25. This is the first statement to execute when operation resumes:

```
25   INPUT ARRAY p_items FROM s_items.*
```

Enter the CONTINUE command or press F4 to resume execution of the program.

The Debugger executes the INPUT ARRAY statement and redisplays the Application screen. The cursor is positioned in the **Stock No.** field of the second row of the screen array, and the program is awaiting input, as shown in Figure 6-18.

Figure 6-18
Resuming Execution of the Program



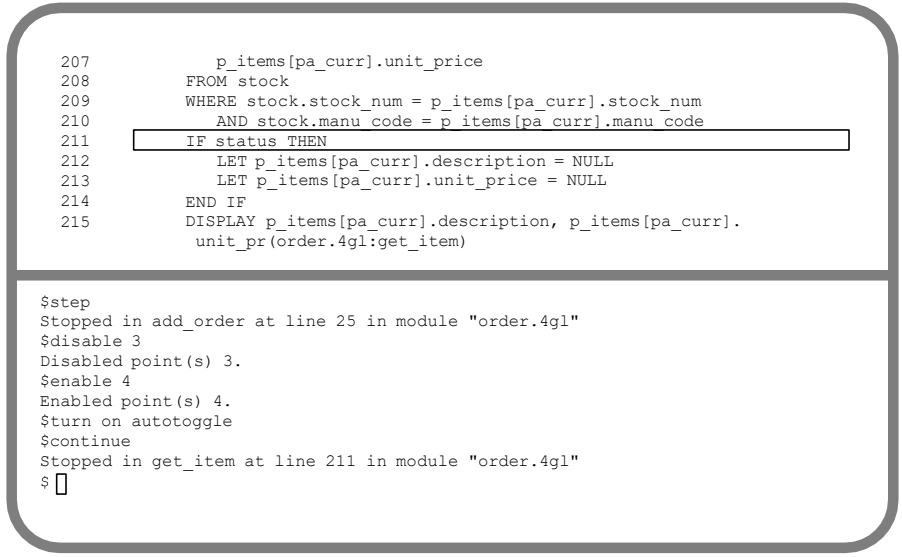
Reaching the Second Breakpoint

You have set a breakpoint whenever the value of **status** is set to `100`. Suspending execution when invalid values are entered allows you to observe the feedback provided to the user and to experiment with different strategies for correcting the data and resetting the value of **status**.

To suspend program execution following a lookup of the **stock** table, you need to enter an invalid set of values into the **Stock No.** and **Code** fields. For example, enter `1` in the **Stock No.** field and enter `ANZ` in the **Code** field.

Because the combination `1` and `ANZ` does not exist in the **stock** table, the value of **status** is set to `100`, and the condition specified in the breakpoint is met. Program execution is suspended, and the Debugger windows appear as shown in Figure 6-19.

Figure 6-19
Reaching the Second Breakpoint



When you set a breakpoint on an IF condition, execution is suspended when the expression evaluates to TRUE. In this example, the value of **status** is set to `100` immediately following execution of the SELECT statement beginning at line 205. The Source window highlights the next statement to execute when program operation resumes. This is line 211:

```
211   IF status THEN
```

The Command window displays the function name, line number, and module name at which the breakpoint occurred. This is line 211 in the function **get_item** in the **order** module.

Resuming Operation Following the Second Breakpoint

To add the current row successfully, one of the entered values must be changed to produce a valid **Stock No.** and **Code** field combination. There are two ways this can be accomplished:

- You can resume execution, changing values as directed by the program. This allows you to observe the program response to invalid values and the feedback it provides to the user.
- You can change the value of one of the variables using LET, and reexecute the function that performs the lookup with the CALL command. This allows you to continue execution following the breakpoint with valid values.

This section illustrates the first of these approaches. The section "Executing a Function Interactively" on page 6-35 illustrates the second approach and shows how to use the Debugger to recall the **get_item** function.

To change field values as directed by the program

1. Enter the CONTINUE command or press F4.

   The Debugger redisplays the Application screen, with the cursor positioned in the **Quantity** field.

2. Enter a value of `10` in this field and press RETURN.

   The appearance of the Application screen when you have made your entry is as shown in Figure 6-20.

Figure 6-20
Entering Invalid Values

```
Enter a stock number or press CTRL-B to scan stock list
Press ESC to write order
------------------------------------------------------------------------------

                              ORDER FORM

------------------------------------------------------------------------------

 Customer Number: 108            Contact Name: Donald            Quinn
    Company Name: Quinn's Sports
         Address: 587 Alvarado
            City: Redwood City      State: CA  Zip Code: 94063
       Telephone: 415-544-8729
------------------------------------------------------------------------------

 Order No:[           ]    Order Date: 09/24/1987     PO Number: JR1447
        Shipping Instructions: overnight delivery
------------------------------------------------------------------------------

 Item No. Stock No.  Code    Description     Quantity   Price        Total
 [    1]  [     5]  [NRG] [tennis racquet] [    12] [  $28.00] [   $336.00]
 [     ]  [     1]  [ANZ] [               ] [   10] [         ] [          ]
 [     ]  [      ]  [   ] [               ] [     ] [         ] [          ]
 [     ]  [      ]  [   ] [               ] [     ] [         ] [          ]
             Running Total including Tax and Shipping Charges:[  $369.60]
A valid stock number, manufacturer, and quantity must all be entered
 ============================================================================
```

The cursor returns to the **Stock No.** field, and the program displays the following message:

```
A valid stock number, manufacturer, and quantity must all be entered
```

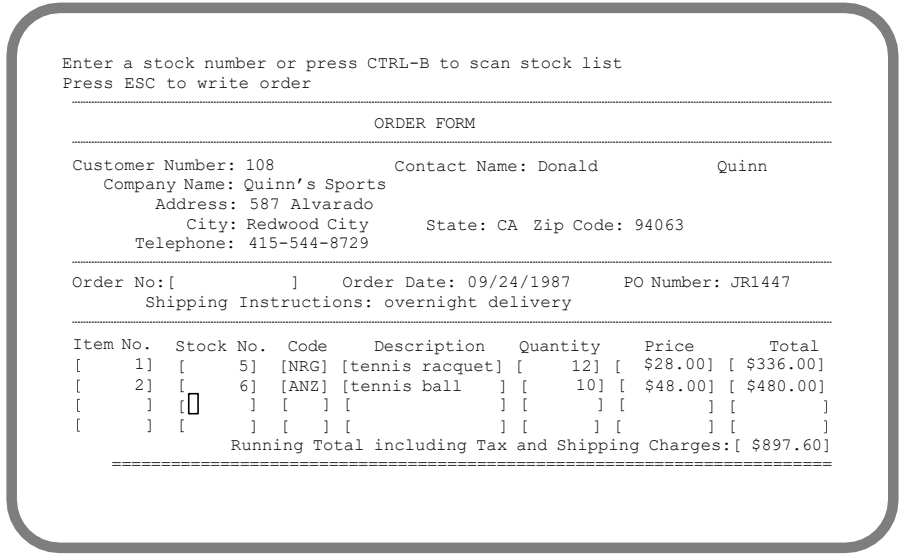
The program prevents you, therefore, from adding a new row to the order until all three values required from the user are correct.

There is a valid item in the **stock** table with **Stock No.** entry of `6` and a **Code** field entry of `ANZ`. To replace the current values with those for this item, you should perform the following steps.

3. Change the value in the **Stock No.** field to `6`, and press RETURN.
4. Press RETURN two more times to move the cursor through the **Code** and **Quantity** fields.

   Because the lookup of the current item is successful, the program displays values in the remaining fields of this row and in the **Running Total including Tax and Shipping Charges** field. It adds this row to the **p_items** program array and positions the cursor in the **Stock No.** field of the third row of the screen array. The appearance of the Application screen after entry of this row is as shown in Figure 6-21.

Figure 6-21
Correcting Values Through the Program



Executing a Function Interactively

This section illustrates using Debugger commands to change an invalid value and to reexecute the **get_item** function.

Enter the same invalid values as previously into the **Stock No.** and **Code** fields. These values are `1` and `ANZ`.

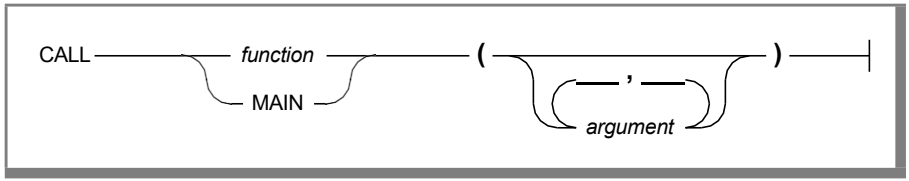
As soon as you press RETURN to leave the **Code** field on this row, the value of **status** is again set to `100`. Because the breakpoint you have set at this condition is still active, program execution is suspended again. Control returns to the Debugger, which redisplays the Source and Command windows as shown in Figure 6-22.

Figure 6-22
Reaching the Second Breakpoint Again

```
  207         p_items[pa_curr].unit_price
  208      FROM stock
  209      WHERE stock.stock_num = p_items[pa_curr].stock_num
  210         AND stock.manu_code = p_items[pa_curr].manu_code
  211      IF status THEN
  212         LET p_items[pa_curr].description = NULL
  213         LET p_items[pa_curr].unit_price = NULL
  214      END IF
  215      DISPLAY p items[pa curr].description, p items[pa curr].unit pr
(order.4gl:get_item)
```

```
$disable 3
Disabled point(s) 3.
$enable 4
Enabled point(s) 4.
$turn on autotoggle
$continue
Stopped in get_item at line 211 in module "order.4gl"
$continue
Stopped in get_item at line 211 in module "order.4gl"
$
```

The CALL Command



You can use the CALL command to execute a function interactively. In the present example, you use the CALL command in conjunction with LET to reexecute the **get_item** function with a different value following the new occurrence of the breakpoint. You must use the parentheses when you call a function. If the function has arguments, you must include them between the parentheses.

You can also use the CALL command to execute a C function that has been linked to your 4GL program from a library. There are, however, some restrictions on the Debugger commands that can be used with C functions. See Appendix B, "Calling C Functions," for procedures and examples of using the Debugger with a 4GL program that calls C functions.

Changing a Value with the LET Command

Prior to recalling the **get_item** function, you want to change one of the entered values so that it identifies a valid row. The current break in program execution was produced when you entered `1` in the **Stock No.** field and `ANZ` in the **Code** field.

There is a valid row in the **stock** table with **Stock No.** `1` and **Code** `HRO`. Therefore, enter the LET command as follows to change the current value of the variable **p_items[pa_curr].manu_code** from `ANZ` to `HRO`:

```
LET  p_items[pa_curr].manu_code  =  "HRO"
```

You must place the values of character variables in quotes when using the LET command.

Entering the CALL Command

You are now ready to reexecute the **get_item** function with the new value.

To reexecute the get_item function

1. Enter the CALL command as follows to reexecute the **get_item** function with the values `1` and `HRO`:

   ```
   CALL  get_item()
   ```

2. Because you have turned the AUTOTOGGLE parameter back to ON, the Debugger switches briefly to the Application screen when the following statement executes:

   ```
   215   DISPLAY p_items[pa_curr].description, p_items[pa_curr].unit_price
   216       TO s_items[sc_curr].description, p_items[sc_curr].unit_price
   ```

   When the **get_item** function terminates, the Source and Command windows reappear as shown in Figure 6-23.

Figure 6-23
Calling the get_item Function

```
207             p_items[pa_curr].unit_price
208          FROM stock
209          WHERE stock.stock_num = p_items[pa_curr].stock_num
210             AND stock.manu_code = p_items[pa_curr].manu_code
211          IF status THEN
212             LET p_items[pa_curr].description = NULL
213             LET p_items[pa_curr].unit_price = NULL
214          END IF
215          DISPLAY p_items[pa_curr].description, p_items[pa_curr]
              .unit_pr(order.4gl:get_item)
```

```
$turn on autotoggle
$continue
Stopped in get_item at line 211 in module "order.4gl"
$continue
Stopped in get_item at line 211 in module "order.4gl"
$let p_items[pa_curr].manu_code = "HRO"
$call get_item()
Return from get_item at line 220
Stopped in get_item at line 211 in module "order.4gl"
$
```

Appearance of the Source Window

When you use the CALL command, the called function executes completely. Program control then returns to the point in the program from which you issued the call.

Because line 211 in the **order** module was the next statement to execute when you entered the command:

```
CALL get_item()
```

this statement is highlighted in the Source window when the **get_item** function terminates.

Appearance of the Command Window

The Debugger outputs the following information in the Command window:

- The line number at which the function terminated. This is line 220.
- The function name, line number, and module name at which execution stopped. This is line 211, which is the next statement to execute when operation resumes.

Resuming Operation After CALL

The first statement to execute when operation resumes is line 211:

```
211   IF STATUS THEN
```

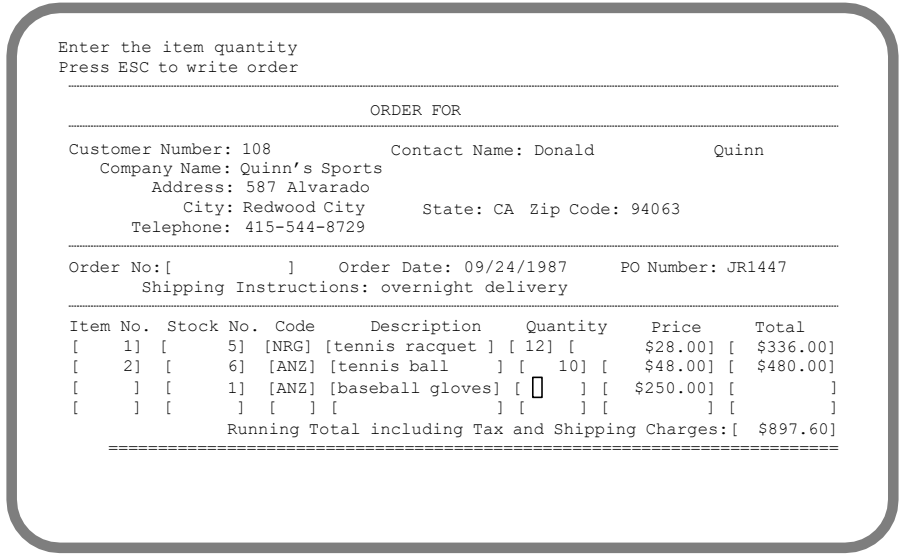
To resume operation

1. Enter the CONTINUE command or press F4 to resume operation with this statement.

   Because you have used the LET command to produce a valid **Stock No.** and **Code** combination, the value of **status** is 0, and no break occurs. The remaining statements of the **get_item** function execute, and control returns to **add_order**.

   When the Debugger redisplays the Application screen, the **Description** and **Price** are displayed for the new item, and the cursor is positioned in the **Quantity** field, as shown in Figure 6-24.

Figure 6-24
Resuming Operation After CALL



2. Enter `6` in this field and press RETURN to add the current row.

As soon as you press RETURN, the cursor leaves the third row of the screen array, and the value `ANZ` in the **Code** field is replaced with the value `HRO`.

Execution of the Tracepoints

In carrying out the debugging steps in this chapter, you have entered three items for a customer order.

To review the tracepoints

1. Press ESC to terminate your order entry and to insert the order and item information into the database.

   The program displays the message `Order added` as the order is added to the database.

   As soon as you press ESC, **add_order** executes the following statement at line 92:

   ```
   92   CALL insert_order()
   ```

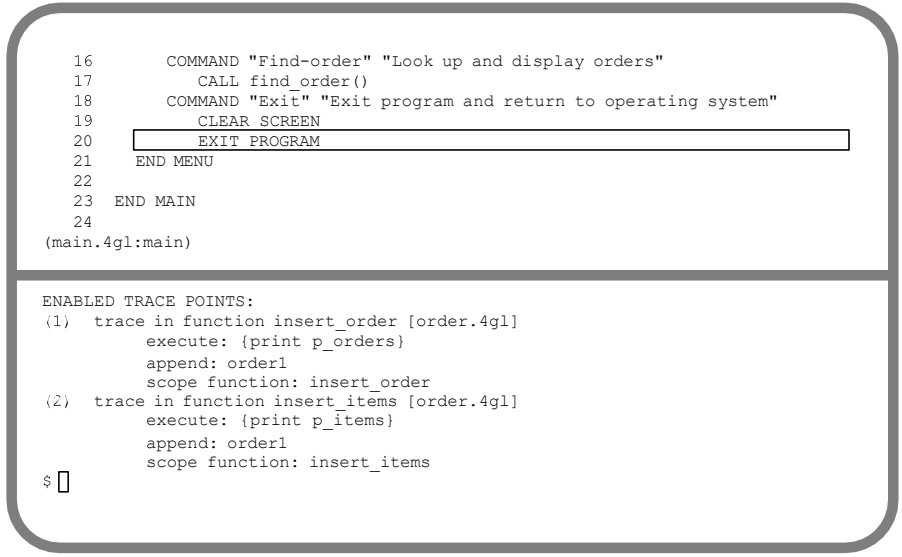
   The **insert_order** function in turn calls the **insert_items** function as follows:

   ```
   110   IF NOT insert_items() THEN
   ```

   At the beginning of this chapter, you set active tracepoints on both the **insert_order** and the **insert_items** functions. Both of these tracepoints generate their output now that order entry is complete.
2. Choose the **Exit** option to terminate the program and return the cursor to the Command window.
3. Use the LIST TRACE command to review these tracepoints.

   Figure 6-25 illustrates the Debugger response to this command:

Figure 6-25
Listing the Tracepoints



Contents of the order1 File

Because you redirected the output of the tracepoints to the file **order1**, no information is displayed in the Command window. Instead, a file with the name **order1** is created in the current directory when the first tracepoint generates output, and the output of both tracepoints is written to it.

The order of the information written to the **order1** file reflects the nesting of the functions **insert_order** and **insert_items**. You can use the exclamation point (!) to escape from the Debugger to the operating system and view this file.

To view the contents of order1

1. Enter an exclamation point followed by the appropriate operating system command (such as **cat**, **more**, or **page**).

   For example:

   ```
   !cat order1
   ```

2. Press any key to return the cursor to the Command window.

   The following example lists the contents of the **order1** file:

```
Enter insert_order() from add_order line 92
global:p_orders  = record
     order_num  =  0
     order_date = 09/24/1987
     po_num = "JR1147      "
     ship_instruct  =  "overnight  delivery
          "
end  record
Enter insert_items() from insert_order line 110
global:p_items  = {
     item_num  =  1
     stock_num = 5
     manu_code = "NRG"
     description = "tennis racquet "
     quantity = 12
     unit_price = $28.00
     total_price = $336.00
     item_num = 2
     stock_num = 6
     manu_code = "ANZ"
     description = "tennis ball       "
     quantity = 10
     unit_price = $48.00
     total_price = $480.00
     item_num = 3
     stock_num = 1
     manu_code = "HRO"
     description = "baseball gloves"
     quantity = 6
     unit_price = $250.00
     total_price = $1500.00
     item_num = (null)
     stock_num = (null)
     manu_code = (null)
     description = (null)
     quantity = (null)
     unit_price = (null)
     total_price = (null)
     item_num = 0
     stock_num = 0
     manu_code = (null)
     description = (null)
```

```
            quantity = 0
            unit_price = (null)
            total_price = (null)
            item_num = 0
            stock_num = 0
            manu_code = (null)
            description = (null)
            quantity = 0
            unit_price = (null)
            total_price = (null)
            item_num = 0
            stock_num = 0
            manu_code = (null)
            description = (null)
            quantity = 0
            unit_price = (null)
            total_price = (null)
            item_num = 0
            stock_num = 0
            manu_code = (null)
            description = (null)
            quantity = 0
            unit_price = (null)
            total_price = (null)
            item_num = 0
            stock_num = 0
            manu_code = (null)
            description = (null)
            quantity = 0
            unit_price = (null)
            total_price = (null)
            item_num = 0
            stock_num = 0
            manu_code = (null)
            description = (null)
            quantity = 0
            unit_price = (null)
            total_price = (null)
        }
        Return (1) from insert_items at line 179
        Return from insert_order at line 119
```

Output of the First Tracepoint

The command **trace insert_order** is responsible for the first and last lines of the file. They record when the **insert_order** function is called and when it ends execution:

```
Enter insert_order() from add_order line 92
   ...
Return from insert_order at line 119
```

The instruction **{ print p_orders }** is responsible for the following lines of output:

```
global:p_orders  = record
     order_num  =  0
     order_date = 09/24/1987
     po_num = "JR1147      "
     ship_instruct = "overnight delivery                          "
end record
```

These lines display the values of the members of the **p_orders** program record when the **insert_order** function was entered. The values for the **order_date**, **po_num**, and **ship_instruct** variables are those entered by you. The value of the **order_num** variable has not yet been assigned when the **insert_order** function is called, and it is initialized to `0`.

Output of the Second Tracepoint

The command **trace insert_items** is responsible for the eighth and the second from the last lines of the file. They record when the **insert_items** function is called and when it ends execution:

```
Enter  insert_items()  from  insert_order  line  110
                    ....
Return  (1)  from  insert_items  at  line  179
```

Because **insert_items** returns the value `1`, or TRUE, to the calling function, this information is listed in the output of the TRACE command as well.

The instruction **{ print p_items }** is responsible for the remaining lines of output to **order1**. These lines display the values in each row of the **p_items** program array when the **insert_items** function was called. The appearance of the first filled row is as follows:

```
item_num  =  1
stock_num = 5
manu_code = "NRG"
description = "tennis racquet "
quantity = 12
unit_price = $28.00
total_price = $336.00
```

The values of the **stock_num**, **manu_code**, and **quantity** variables for each row are entered by you. The remaining values for each row are looked up or calculated by the program. The values of the members of the rows that are not filled are initialized to `0` or NULL.

Chapter Summary

You can define tracepoints and breakpoints without enabling them. This makes it possible to define multiple points at the beginning of a debugging session or in a **.4db** file and to enable, or activate, them as needed.

The STEP command gives you the option of executing all the statements in a function as a unit or as individual steps. When you use STEP to execute a function call, all the statements in the function are treated as a unit. When you use STEP with the INTO option, the Debugger highlights the first executable statement in the function and allows you to execute the statements in the function individually.

You can use the CALL command to execute a function interactively. Using the CALL command can save you time by allowing you to go immediately to a program segment that you want to examine or want to reexecute with different values. When you use the CALL command, program control returns to the point from which you issued the call.

Chapter

7

Analyzing Runtime Errors in the cust_order Program

In This Chapter

This chapter uses the **cust_order** program introduced in Chapter 5, "A Multi-Module Program: cust_order," to illustrate operation of the Debugger when fatal errors occur. Two intentional bugs have been coded into the program for this purpose. This chapter introduces several new Debugger commands and shows how they can be used to easily diagnose the cause of these two common runtime errors. It then shows you how to make the appropriate corrections and how to recompile the **cust_order** program. The following topics are covered in this chapter:

- The appearance of the Source and Command windows when a fatal error occurs
- The use of the VARIABLE command to display the definition of a program variable
- The use of the WHERE command to list the functions that have been called leading up to the current INFORMIX-4GL statement
- Use of the Programmer's Environment to correct and recompile the program

Encountering Runtime Errors

Under certain conditions the **cust_order** program produces runtime errors that abort the operation of the program.

Fatal Errors When Running a Program

When an 4GL program is running outside the Debugger, a fatal error by default causes program execution to terminate. An error number and message appear on the screen.

You can specify the action your program should take if a runtime error occurs by using the WHENEVER ERROR statement with the options CONTINUE, CALL, and GOTO. (See the *INFORMIX-4GL Reference* for more information on the WHENEVER ERROR statement.)

Fatal Errors When Debugging a Program

When a fatal error occurs in a program that is running through the Debugger, the following actions take place:

- Control returns immediately to the Debugger, which displays the Source and Command windows.
- The Source window highlights the statement at which execution terminated.
- The Command window displays the information that a fatal error has occurred, as well as the function name, line number, and module name at which the error occurred.

If you have included WHENEVER ERROR statements in your program, the Debugger executes these statements if possible rather than terminating execution. If the program cannot recover from the error, the Debugger takes the action described above.

You can perform diagnostic tests following a fatal error to determine its cause. You can then rerun the program with the RUN command or recall a particular function with the CALL command. You cannot resume program execution following a fatal error with the CONTINUE or STEP command.

Starting the Session

You should initiate the debugging session as in the previous two chapters.

To start the debugging session

1. Choose the **Debug** option from the **PROGRAM** menu.
2. Choose the **cust_order** program.

   If you have exited from the Debugger but have not exited from the Programmer's Environment since working with the examples in Chapter 6, "Tracing Logic of the cust_order Program," the two tracepoints and two breakpoints that you defined in that chapter are automatically restored. Use the DISABLE command with the option ALL to disable any active tracepoints or breakpoints before proceeding with the examples in this chapter.
3. When the Debugger windows appear on the screen, enter `run` or press F5 to begin operation, as shown in Figure 7-1.

Figure 7-1
Starting the Session

Fatal Error #1: Exceeding Terminal Display Limits

The **cust_order** program uses the **query_customer** function to retrieve a customer for whom to place an order. This function is called as soon as you choose the **Add-order** option from the **cust_order** program menu. The **add_order** function checks to see if the value returned by **query_customer** is `1` (TRUE), or `0` (FALSE), and assigns this value to the module variable **query_stat.** The following figure lists the block of code in the **add_order** function where the call to **query_customer** is made and where the value of **query_stat** is assigned:

```
7    FUNCTION  add_order()
8       DEFINE  pa_curr,  s_curr  INTEGER
9
10      LET  query_stat  =  query_customer()
11      IF  query_stat  =  0  THEN
12        RETURN
13      END  IF
14      DISPLAY  BY  NAME  p_customer.*  ATTRIBUTE(CYAN)
```

If a customer both exists and is selected, **query_customer** returns a value of TRUE. It returns a value of FALSE under either of the following conditions:

- No customer exists with the specified search criteria.
- No customer is selected through the **Select** menu option.

In either of these cases, the program is designed to display a message and to return the cursor to the program menu so that you can make another selection.

If no customer exists, the program executes the following lines of the **query_customer** function:

```
126   IF  NOT  exist  THEN
127      CLEAR  FORM
128      CALL  mess("No  customer  satisfies  query",  mrow)
129      LET  p_customer.customer_num  =  NULL
130      RETURN  (FALSE)
131   END  IF
```

If no customer is selected, the program executes the following lines of the **query_customer** function:

```
132   IF NOT chosen THEN
133      CLEAR FORM
134      CALL mess("No selection made", mrow)
135      LET p_customer.customer_num = NULL
136      RETURN (FALSE)
137   END IF
```

In both cases, **query_customer** calls the **mess** function. It passes two arguments to the **mess** function:

- A character string that determines the message to display
- The variable **mrow**, which determines at which row on the terminal screen the message is to display

The contents of the **mess** function are as follows:

```
27   FUNCTION mess(str, mrow)
28       DEFINE str CHAR(80),
29              mrow SMALLINT
30
31       DISPLAY " ", str CLIPPED AT mrow,1
32       SLEEP 3
33       DISPLAY "" AT mrow,1
34   END FUNCTION
```

Producing the First Error

Both situations in which the **query_customer** function returns FALSE result in a fatal error.

To produce the first error

1. Choose the **Add-order** option from the **cust_order** program menu.
2. Enter the invalid search criterion `Customer Number > 200`, as shown in Figure 7-2, and press ESC.

Figure 7-2
Entering Invalid Search Criteria

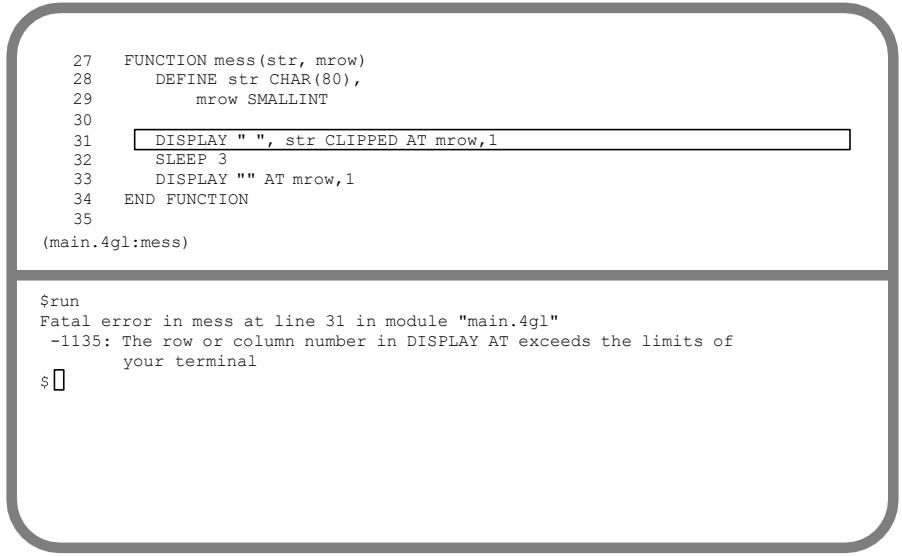
```
Enter criteria for selection
------------------------------------------------------------------------------

                                ORDER FORM
------------------------------------------------------------------------------
 Customer Number:[>200       ]   Contact Name:[                ][          ]
    Company Name:[                         ]
         Address:[                         ][                    ]
            City:[                       ] State:[   ] Zip Code:[      ]
       Telephone:[                        ]
------------------------------------------------------------------------------
 Order No:[           ]    Order Date:[           ]    PO Number:[         ]
        Shipping Instructions:[                                             ]
------------------------------------------------------------------------------
 Item No.  Stock No.  Code      Description     Quantity     Price        Total
 [       ] [       ] [    ] [                ] [       ] [          ] [      ]
 [       ] [       ] [    ] [                ] [       ] [          ] [      ]
 [       ] [       ] [    ] [                ] [       ] [          ] [      ]
 [       ] [       ] [    ] [                ] [       ] [          ] [      ]
                  Running Total including Tax and Shipping Charges:[       ]
   ============================================================================
```

Because there are no customers in the **stores7** database with customer numbers greater than 200, the value of **exist** is set to FALSE. If **exist** is FALSE, the program should display the following message and return the cursor to the program menu:

```
No customer satisfies query
```

Instead, entering the invalid search criterion produces a fatal error. Program execution terminates, and the Source and Command windows redisplay as shown in Figure 7-3.

Figure 7-3
Exceeding Terminal Display Limits



When a fatal error occurs, the Debugger highlights in the Source window the line at which the fatal error occurred. In the present example, this is line 31 in the **main** module:

```
31   DISPLAY " ", str CLIPPED AT mrow,1
```

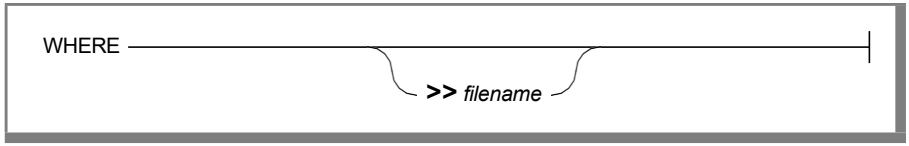
The Debugger outputs to the Command window the function name, line number, and module name at which the error occurred. It also displays the error number and corresponding error message. In the present example, this is error number -1135:

```
Fatal error in mess at line 31 in module "main.4gl"
    -1135: The row or column number in DISPLAY AT exceeds the limits of
    your terminal.
```

The DISPLAY AT statement displays a message at a specific row and column of the terminal screen. This error indicates that the value of **mrow**, which determines the row at which a message is to display on the screen, is not valid.

The WHERE Command

The following diagram describes the syntax of the WHERE command.

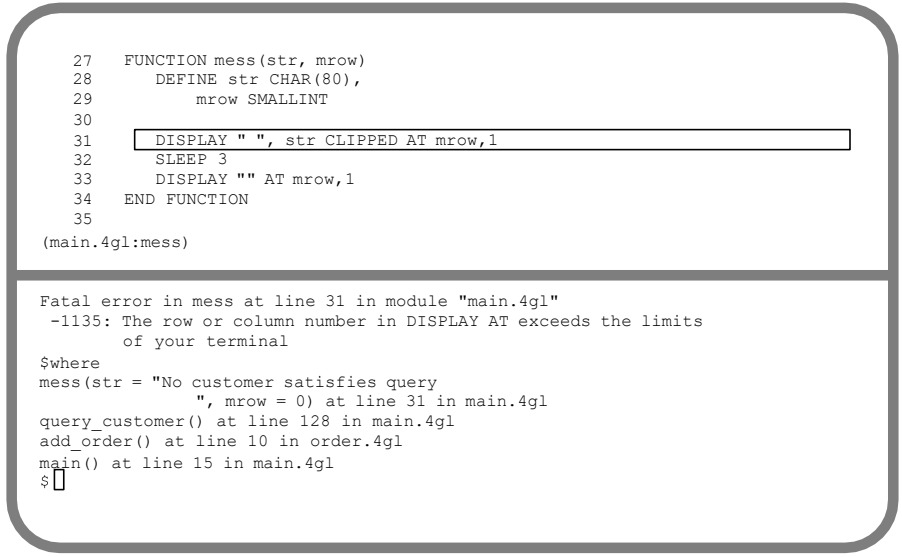


You are now ready to perform diagnostic tests to determine the cause of the error. You could, for example, use the PRINT command with which you are already familiar to display the value of the variable **mrow** and to verify that the number assigned to it is not a valid row number for the terminal. In diagnosing the cause of the error, however, it would also be useful to know how **mrow** received its value.

You can use the WHERE command to display all of the functions that have been called leading up to the current 4GL statement. If any of the functions are called with parameters, the values of these parameters are displayed as well.

Enter `where` to display the path that the program has taken in arriving at line 31. Figure 7-4 illustrates your entry of the WHERE command and the Debugger output.

Figure 7-4
The WHERE Command



Output of the WHERE Command

The Debugger lists the functions called leading up to the current statement in reverse chronological order of their execution. In the present example, the current function is **mess**. You see that this function received two parameters when it was called and that the calling function was **query_customer**. The first parameter, **str**, has the value:

```
"No customer satisfies query"
```

The second parameter, **mrow**, has a value of `0`. This is the value that produced the fatal error because `0` is not a valid coordinate for the DISPLAY AT statement.

Viewing the Calling Function in the Source Window

You see also from the output of the WHERE command that these values were determined in line 128 of the calling function **query_customer**. This function is in the module **main**, which is currently displayed in the Source window.

To view the calling function

1. Enter the VIEW command to move the cursor to the Source window and examine this module.
2. Move to line 128 of the module by typing `128`, followed by RETURN.

   The Source window displays the lines of code shown in Figure 7-5.

Figure 7-5
Examining the Calling Function in the Source Window

```
  124
  125       CALL clear_menu()
  126       IF NOT exist THEN
  127          CLEAR FORM
  128          CALL mess("No customer satisfies query", mrow)
  129          LET p_customer.customer_num = NULL
  130          RETURN (FALSE)
  131       END IF
  132       IF NOT chosen THEN
(main.4gl:query_customer)
```

```
 -1135: The row or column number in DISPLAY AT exceeds the limits of
        your terminal
$where
mess(str = "No customer satisfies query
               ", mrow = 0) at line 31 in main.4gl
query_customer() at line 128 in main.4gl
add_order() at line 10 in order.4gl
main() at line 15 in main.4gl
$view
```

3. Use UP ARROW and DOWN ARROW or the cursor movement CTRL keys described in Chapter 2, "Getting Started with the Debugger," to scroll the **query_customer** function through the Source window.

   You can observe that while the variable **mrow** is defined at line 65 of this function and is passed as a parameter to the **mess** function, it is never assigned a value in **query_customer**. Also, because **mrow** does not appear as an argument of **query_customer**, it cannot receive its value from outside the function.

   The programmer, therefore, has defined this variable in the program with the intention of providing flexibility in the coordinates of the DISPLAY AT statement but has failed to assign it a value.
4. Press the **Interrupt** key when you are finished to return the cursor to the Command window.

A Possible Solution

In order for the **cust_order** program to work correctly, you must assign a value to **mrow**. There are several ways that this might be accomplished. One solution is to carry out the following activities:

- Make **mrow** an argument of the **query_customer** function as follows:

```
61   FUNCTION query_customer(mrow)
```

- Pass a value for **mrow** when the **query_customer** function is called by **add_order**, as follows:

```
10   LET query_stat = query_customer(2)
```

In this way, the value of **mrow** is set to `2` when **query_customer** is called, and the resulting coordinates of the DISPLAY AT statement at line 31 are now `2,1`:

```
31   DISPLAY " ", str CLIPPED AT mrow, 1
```

When this change is made, the messages:

```
No customer satisfies query
```

and:

```
No selection made
```

appear, beginning in the first column of the second line of the terminal screen.

"Correcting the Program" on page 7-22 provides instructions on making these changes to the **cust_order** program modules and on recompiling the program.

Fatal Error #2: Exceeding Array Bounds

In the previous chapter, you placed an item on a customer order by making entries in the **Stock No.** and **Code** fields on the form. The **cust_order** program is designed to give you the option of viewing a list of available stock items in a window when placing an order and of making a selection directly from this list. An error occurs, however, when you use this option.

To produce fatal error #2

1. Enter the RUN command or press F5 to restart operation of the Debugger.
2. Choose the **Add-order** option from the **cust_order** program menu.
3. Enter search criteria for customer 109, Jane Miller, and press ESC.
4. Choose the **Select** option from the **BROWSE** menu to select this customer.
5. Enter sample values for the fields labeled **Order Date**, **PO Number**, and **Shipping Instructions**, and press RETURN.

Figure 7-6 illustrates the appearance of the screen with some sample order information for Jane Miller.

Figure 7-6
Entering Order Information

```
Enter a stock number or press CTRL-B to scan stock list
Press ESC to write order
------------------------------------------------------------------------------
                                ORDER FORM
------------------------------------------------------------------------------
 Customer Number:[109         ]  Contact Name:[Jane           ][Miller    ]
    Company Name:[Sport Stuff          ]
         Address:[Mayfair Mart         ][7345 Ross Blvd.      ]
            City:[Sunnyvale      ] State:[CA] Zip Code:[94086]
       Telephone:[408-723-8789           ]
------------------------------------------------------------------------------
 Order No:[        ]    Order Date:[07/08/1987]    PO Number:[J90681    ]
        Shipping Instructions:[fed ex                                      ]
------------------------------------------------------------------------------
 Item No.  Stock No.  Code    Description     Quantity     Price     Total
 [       ] [       ] [    ] [              ] [       ] [          ] [       ]
 [       ] [       ] [    ] [              ] [       ] [          ] [       ]
 [       ] [       ] [    ] [              ] [       ] [          ] [       ]
 [       ] [       ] [    ] [              ] [       ] [          ] [       ]
                Running Total including Tax and Shipping Charges:[        ]
   ============================================================================
```

The listing of stock items to a window, which produces the fatal error, is implemented by means of an ON KEY clause within the INPUT ARRAY block. This block of code is initiated by the following statement:

```
25   INPUT ARRAY p_items FROM s_items.*
```

The 4GL statements that execute if the user presses CTRL-B in either of the designated fields are as follows:

```
40  ON KEY (CONTROL-B)
41     IF INFIELD(stock_num) OR INFIELD(manu_code) THEN
42         LET pa_curr = arr_curr()
43         LET s_curr = scr_line()
44         CALL fetch_stock()
45         CALL get_stock() RETURNING
46            p_items[pa_curr].stock_num, p_items[pa_curr].manu_code,
47            p_items[pa_curr].description, p_items[pa_curr].unit_price
48         DISPLAY p_items[pa_curr].stock_num
49            TO s_items[s_curr].stock_num
50         DISPLAY p_items[pa_curr].manu_code
51            TO s_items[s_curr].manu_code
52         DISPLAY p_items[pa_curr].description
53            TO s_items[s_curr].description
54         DISPLAY p_items[pa_curr].unit_price
55            TO s_items[s_curr].unit_price
56         NEXT FIELD quantity
57     END IF
```

If you press CTRL-B while the cursor is in either the **stock_num** field or the **manu_code** field, the program calls the **fetch_stock** function. This function declares the cursor **stock_list** and retrieves rows into the program array **p_stock** by means of a FOREACH loop. The contents of the **fetch_stock** function are as follows:

```
46  FUNCTION fetch_stock()
47
48      DECLARE stock_list CURSOR FOR
49          SELECT stock_num, manufact.manu_code,
50                 manu_name, description, unit_price, unit_descr
51          FROM stock, manufact
52          WHERE stock.manu_code = manufact.manu_code
53          ORDER BY stock_num
54      LET stock_cnt = 1
55      FOREACH stock_list INTO p_stock[stock_cnt].*
56          LET stock_cnt = stock_cnt + 1
57      END FOREACH
58      LET stock_cnt = stock_cnt - 1
59  END FUNCTION
```

After the **fetch_stock** function executes, **add_order** calls the **get_stock** function. This function opens the **stock_w** window with the **stock_sel** form, and uses a DISPLAY ARRAY statement to display the stock items within this window. The contents of the **get_stock** function are as follows:

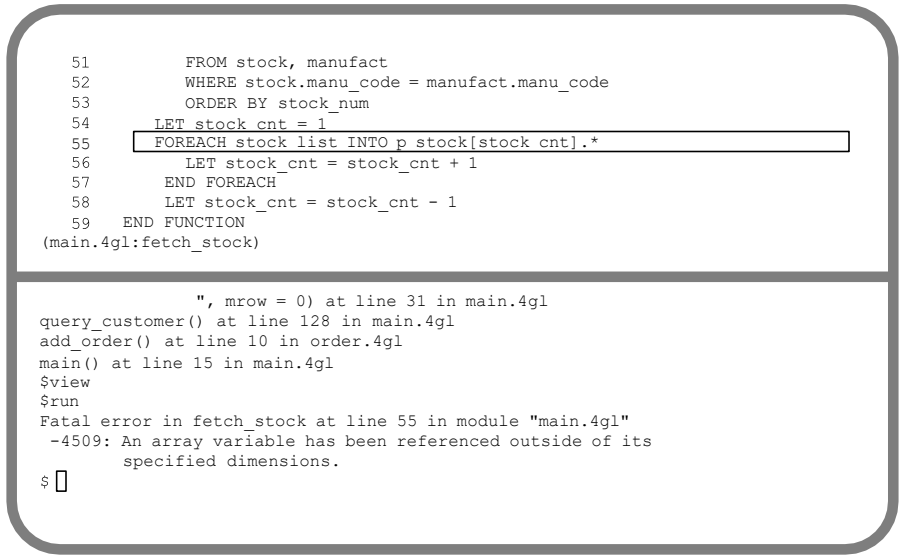
```
183  FUNCTION get_stock()
184      DEFINE idx integer
185
186      OPEN WINDOW stock_w AT 7, 3
186         WITH FORM "stock_sel"
187        ATTRIBUTE(BORDER, YELLOW)
188      CALL set_count(stock_cnt)
189      DISPLAY " Use cursor using F3, F4, and arrow keys; press ESC ",
190         "to select a stock item" AT 1,1
192      DISPLAY ARRAY p_stock TO s_stock.*
193      LET idx = arr_curr()
194      CLOSE WINDOW stock_w
195      RETURN p_stock[idx].stock_num, p_stock[idx].manu_code,
196         p_stock[idx].description,  p_stock[idx].unit_price
197  END FUNCTION
```

See Appendix C, "Sample Programs," for more information on these functions.

Producing the Second Error

Enter CTRL-B to produce the second fatal error. Control returns to the Debugger, which displays the Source and Command windows as shown in Figure 7-7.

Figure 7-7
Exceeding Array Bounds



When a fatal error occurs, the Source window highlights the statement at which program execution terminated. In this example, it is line 55 of the **main** module:

```
55  FOREACH  stock_list  INTO  p_stock[stock_cnt].*
```

The Debugger outputs to the Command window the function name, line number, and module name at which the error occurred. It also displays the error number and corresponding error message. In the present example, this is error number -4509:

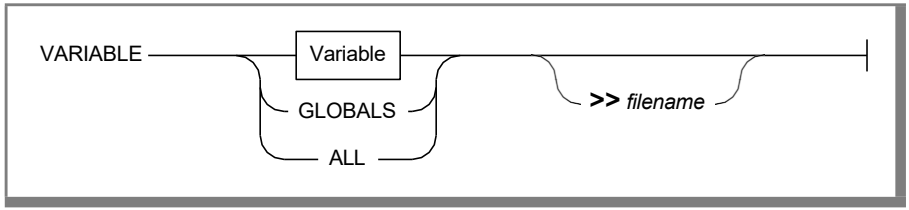
```
Fatal error in fetch_stock at line 55 in module "main.4gl"
   -4509 An array variable has been referenced outside of its
   dimensions.
```

This error indicates that the index for the array was not initialized correctly, or that the cursor attempted to retrieve a row from the database after all the rows in the **p_stock** array had been filled.

The following two sections illustrate the use of the VARIABLE and PRINT commands to quickly determine the cause of this error.

The VARIABLE Command

The following diagram illustrates the syntax of the VARIABLE commands.



The VARIABLE command provides a convenient way to see the declaration of program variables. Use it to avoid having to search the source module or modules for the DEFINE statements associated with particular variables.

For example, enter the following command to display the record members, data types, and, most importantly, the number of elements of the **p_stock** program array:

```
variable  p_stock
```

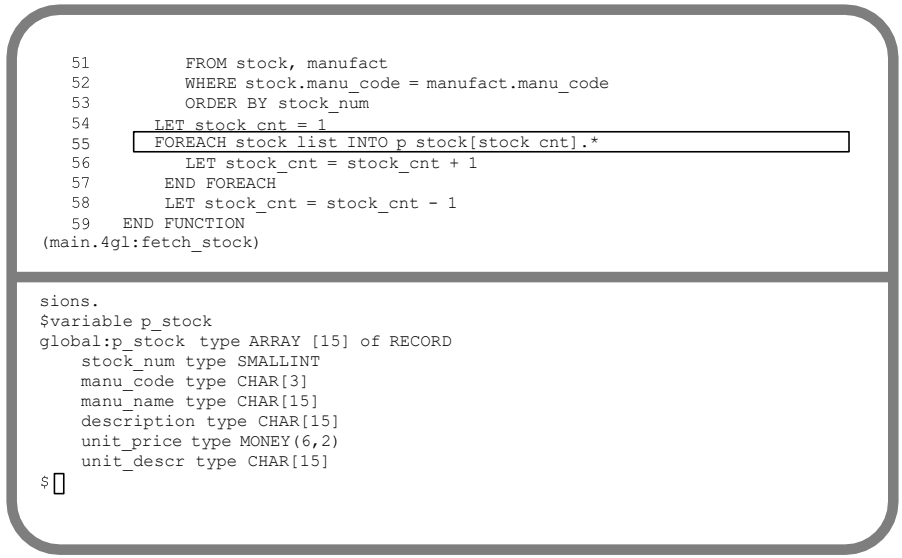
The VARIABLE command outputs the following information to the Command window or, optionally, to a file:

- The data type of a simple program variable
- The record members and data types of a program record
- The members, data types, and number of elements of a program array

You can use the VARIABLE command with the GLOBALS option to display the declarations of all global and module variables in the current function. If you specify the ALL option, the VARIABLE command displays the declarations of all global, module, and local variables in the current function. If you do not specify an option or variable name, the VARIABLE command displays the declaration of all local variables in the current function. See Chapter 9, "The Debugger Commands," for more information on the VARIABLE command and its options.

Figure 7-8 illustrates the output of the VARIABLE command in the Command window.

Figure 7-8
Output from the VARIABLE Command



You see from this example that **p_stock** was defined as an array of 15 elements.

The PRINT Command

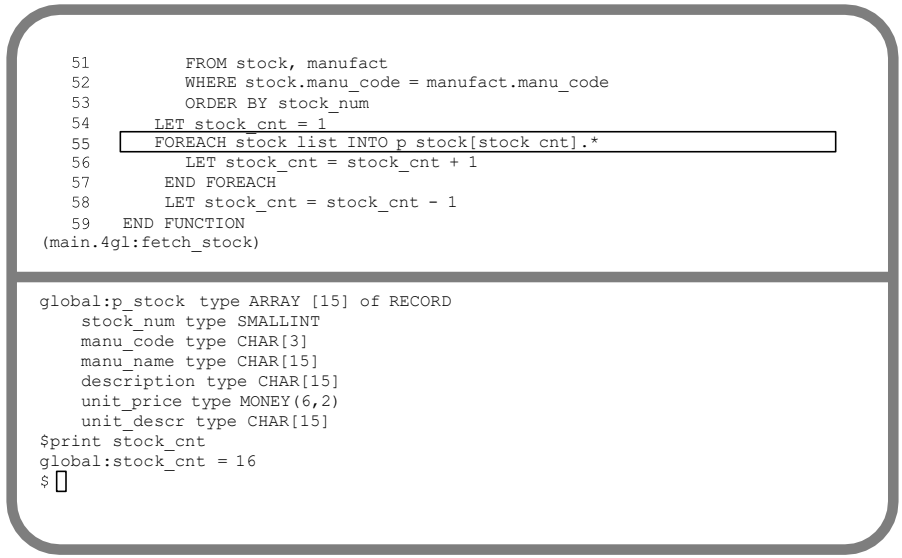
You are also interested in knowing the value of the **stock_cnt** variable, which provides the subscript for the array **p_stock**. You can use the PRINT command with which you are already familiar to display the value of this variable.

Enter the PRINT command as follows to display the current value assigned by the program to **stock_cnt**:

```
print  stock_cnt
```

Figure 7-9 illustrates the output of the PRINT command to the Command window.

Figure 7-9
The PRINT Command



You see from this command that the value assigned to **stock_cnt** when program execution terminated was 16. This produced the fatal error because the number of elements allotted for the **p_stock** program array is 15.

A Possible Solution

To prevent this error from reoccurring any time the number of stock items in the database exceeds 15, you should amend the **fetch_stock** function as follows to include a test for the upper limit of the array:

```
IF stock_cnt > 15 THEN
   EXIT FOREACH
END IF
```

This test should occur right after the value of **stock_cnt** is incremented in line 56:

```
56   LET stock_cnt = stock_cnt + 1
```

Following is the complete text of the **fetch_stock** function with these lines incorporated:

```
FUNCTION fetch_stock()

    DECLARE stock_list CURSOR FOR
    SELECT stock_num, manufact.manu_code,
        manu_name, description, unit_price, unit_descr
        FROM stock, manufact
        WHERE stock.manu_code = manufact.manu_code
        ORDER BY stock_num
    LET stock_cnt = 1
    FOREACH stock_list INTO p_stock[stock_cnt].*
        LET stock_cnt = stock_cnt + 1
        IF stock_cnt > 15 THEN
            EXIT FOREACH
        END IF
    END FOREACH
    LET stock_cnt = stock_cnt - 1
END FUNCTION
```

You can also increase the size of the program array because you know that the number of stock items in the database exceeds 15. However, you should include a similar test to prevent array bounds from being exceeded in the future as new items are added to the **stock** table.

The next section shows you how to correct both of the fatal errors examined in this chapter and how to recompile the **cust_order** program.

Use the EXIT command or press F9 to exit from the Debugger and return to the **PROGRAM** menu.

Correcting the Program

To correct the two errors, you need to make two modifications to the **main** module, and one modification to the **order** module.

To make the corrections to the main module

1. Choose the **Exit** option to return to the **INFORMIX-4GL** menu.
2. Choose the **Module** option from the **INFORMIX-4GL** menu.
3. Choose the **Modify** option from the **MODULE** menu.
4. Choose **main** as the module to modify.

   The first correction involves modifying the code of the **fetch_stock** function to test for the boundaries of the **p_stock** array.
5. Use the system editor to move to line 56 of this module:

   ```
   56   LET stock_cnt = stock_cnt + 1
   ```
6. Insert the next three lines immediately following this statement:

   ```
   57   IF stock_cnt > 15 THEN
   58      EXIT FOREACH
   59   END IF
   ```

   The second correction to this module involves modifying the **query_customer** function so that it is called with **mrow** as an argument. This is the first step required to correct the problem of exceeding the terminal display limits.
7. Use the system editor to move to what is now line 64:

   ```
   64   FUNCTION query_customer()
   ```
8. Make the variable **mrow** an argument of the **query_customer** function as follows:

   ```
   64   FUNCTION query_customer(mrow)
   ```
9. Save the file using the command appropriate to your system editor.
10. Choose the option **Save-and-Exit** when the **MODIFY MODULE** menu appears on the screen.

Correcting the order Module

A second change is required to correct the problem of exceeding the terminal display limits. This change involves modifying the **add_order** function so that **query_customer** is called with an argument that provides the value for **mrow**. The **add_order** function is located in the **order** module.

To access the order module

1. Choose the **Modify** option from the **MODULE** menu.
2. Choose the **order** module.
3. Use the system editor to move to line 10:

```
10   LET query_stat = query_customer()
```

4. Modify this line as follows to pass `2` as the argument to the **query_customer** function:

```
10   LET query_stat = query_customer(2)
```

5. Save the file using the command appropriate to your system editor.
6. Choose the **Save-and-Exit** option when the **MODIFY MODULE** menu appears on the screen, as in the previous example.

Recompiling the Program

You are now ready to recompile the **cust_order** program, incorporating your corrections.

To recompile the program

1. Choose the **Program_Compile** option from the **MODULE** menu.
2. Choose the **cust_order** program.

   The **Program_Compile** option recompiles only those modules that have been altered since the last compilation. The name of each module you have modified is displayed on the screen as the module compiles. Figure 7-10 illustrates the appearance of the screen when the second module has begun the process of recompilation.

Figure 7-10
Recompiling the cust_order Program

```
COMPILE PROGRAM >>
Choose a program with arrow keys or enter a name, and press RETURN.

                                                    Press CTRL-W for Help
Compiling Informix-4GL sources:
                main.4gl
                order.4gl


Compilation in progress...please wait.
```

When the process is complete, you should see the message:

```
Program successfully compiled.
```

3. If errors are discovered in the course of the compilation, choose the **Modify** option to correct the affected module.

Verifying the Corrections

You should rerun the program to verify that the errors have been corrected. Rerun the program by choosing either the **Run** or the **Debug** option from either the **MODULE** menu or the **PROGRAM** menu.

To verify that you have corrected the problem of exceeding the terminal display limits

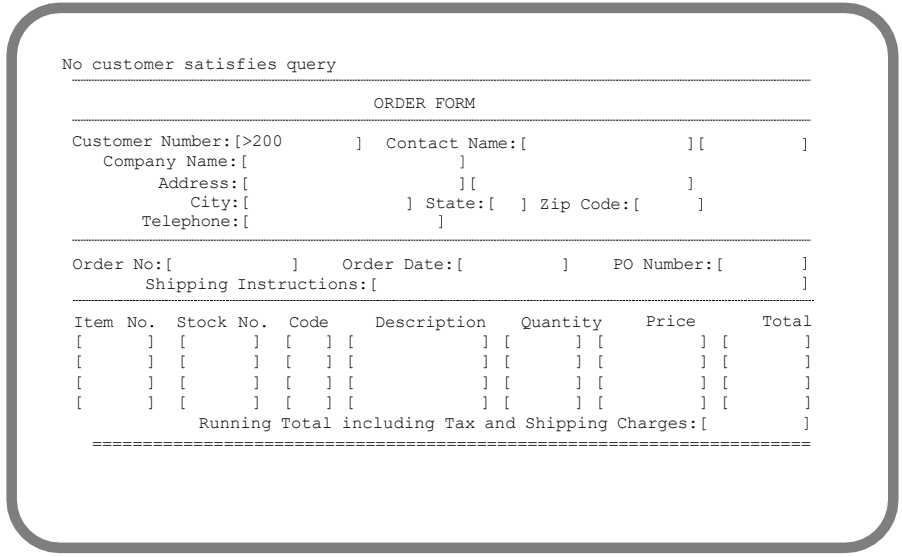
1. Choose the **Add-Order** option from the **cust_order** program menu.
2. Enter the invalid search criterion that you entered previously to produce the problem:

   ```
   Customer Number [>200   ]
   ```

   You see from Figure 7-11 that the program correctly displays the message:

   ```
   No customer satisfies query
   ```

Figure 7-11
Verifying the Correction of the First Error



To verify that you have corrected the problem of exceeding array bounds

1. Choose the **Add-order** option from the program menu.
2. Enter search criteria for one or more customers.
3. Choose the **Select** option to select a customer for whom to place an order.
4. Enter sample order information.
5. Press CTRL-B to display the list of valid stock items to a window on the screen.

You see from Figure 7-12 that the window now opens, and the stock items appear.

Figure 7-12
Verifying the Correction of the Second Error

```
Enter a stock number or press CTRL-B to scan stock list
Press ESC to write order
------------------------------------------------------------------------------

                                ORDER FORM

------------------------------------------------------------------------------

For cursor use F3, F4, and arrow keys; press ESC to select a stock item

     1  HRO    Hero              baseball gloves     $250.00  10 gloves/case
     1  HSK    Husky             baseball gloves     $800.00  10 gloves/case
     1  SMT    Smith             baseball gloves     $450.00  10 gloves/case


------------------------------------------------------------------------------

Item No.  Stock No.  Code    Description    Quantity     Price       Total
[       ] [       ] [    ] [              ] [       ] [          ] [        ]
[       ] [       ] [    ] [              ] [       ] [          ] [        ]
[       ] [       ] [    ] [              ] [       ] [          ] [        ]
[       ] [       ] [    ] [              ] [       ] [          ] [        ]
               Running Total including Tax and Shipping Charges: [        ]
  ============================================================================
```

Chapter Summary

When a 4GL program is running through the Debugger, a fatal error causes control to return to the Debugger. The Source window highlights the line at which program execution terminated. The Command window displays the error number and message, as well as the line number, function name, and module name at which the error occurred. The Debugger makes it easy to diagnose the cause of fatal errors by allowing you to work with a program even after it has aborted.

The VARIABLE command allows you to display the declaration of program variables. The WHERE command allows you to display the functions that have been called to arrive at the current 4GL statement. This command also displays the parameters, if any, with which the functions were called.

To correct errors in a multi-module program, you must modify the individual modules and recompile the program.

This chapter concludes the tutorial section of this manual. You now have a working knowledge of the principal Debugger commands and capabilities, and you should be able to use the Debugger productively with your own programs. The remainder of this manual constitutes a reference guide. It provides additional and detailed information on using the Debugger, as well as the complete syntax of all the Debugger commands.

Chapter

8

The Debugging Environment

In This Chapter

This chapter describes the user interface of the Debugger. Information presented here will help you to control features of your debugging environment. This chapter includes the following procedures:

- Compiling and specifying the INFORMIX-4GL application program
- Specifying the source file search path
- Assigning keyboard aliases
- Using the Debugger screens and windows
- Specifying breakpoints and tracepoints
- Saving debugging environment parameters in a file

Debugger Screens and Parameters

Part of this chapter describes the visual interface of the Debugger, whose screens and windows can display 4GL program input, output, and source code, as well as Debugger output and error messages. Additional displays support Debugger help messages and output from operating system commands.

Display/Window	Information Displayed
Command window	Debugger commands, error messages
Source window	Application program source code
Application screen	Output from the current 4GL program
Help screen	Debugger help messages
Operating system display	Output from operating system commands

Another section identifies the terminal display parameters and the Debugger commands that can specify their values.

Parameter	Controlling Command
AUTOTOGGLE	TURN ON/OFF
DISPLAYSTOPS	TURN ON/OFF
EXITSOURCE	TURN ON/OFF
PRINTDELAY	TURN ON/OFF
SOURCETRACE	TURN ON/OFF
TIMEDELAY SOURCE	TIMEDELAY
TIMEDELAY COMMAND	TIMEDELAY
SOURCE LINES	GROW
COMMAND LINES	GROW
APPLICATION DEVICE	APPLICATION DEVICE

Another section identifies the commands to establish and control breakpoints and tracepoints. It presents and explains many examples of the BREAK and TRACE commands and summarizes restrictions on breakpoints.

This chapter also describes Debugger commands to display the current values of all the debugging environment parameters, to save them in a file, or to restore them from a file.

Command	Purpose
ALIAS	Sets or displays function keys and aliases
LIST	Displays current parameters on the screen
READ	Replaces current parameters with file values
USE	Sets or displays the source file search path
WRITE	Saves current values in a disk file

Another section describes the use of **.4db** files to establish default user interface parameters or to replace the current values.

File	Purpose
$INFORMIXDIR/etc/init.4db	Sets systemwide defaults
$HOME/init.4db	Sets defaults for a specific login account
***program*.4db**	Sets defaults for a specific program
***other*.4db**	Sets new current values (with READ or as a command-line specification)

Most of the commands and Debugger features that are described in this chapter have been mentioned in earlier chapters of this manual.

See also the next chapter, which describes the Debugger command language and presents the complete syntax of all the Debugger commands.

The Debugging Process

The INFORMIX-4GL Interactive Debugger is a source-language debugger for developing 4GL programs. Its advanced multitasking features allow you to choose from a broad range of debugging activities. It provides an environment in which you can perform the following tasks:

- Begin or suspend execution of a 4GL application program.
- Monitor screen output from the 4GL program.
- Provide keyboard input to the 4GL program.
- Display the currently executing 4GL statements.
- Enter commands to control 4GL program execution.
- Display 4GL and Debugger diagnostic error messages.
- Evaluate or modify 4GL program variables.

You can invoke the Debugger from the menu system of the Programmer's Environment, or directly from the operating system prompt. Before you can use the Debugger to analyze a 4GL program, you must first take the following steps:

1. Create or modify a **.4gl** source file.
2. Compile the source file into a **.4go** p-code file.
3. Combine multiple **.4go** modules into a single **.4gi** file.
4. Invoke the Debugger, specifying a 4GL program.

Step 3 is not required if your program has only one module.

The Debugger does not modify your source files. If you discover errors in the logic of a **.4gl** source file, the Debugger will help you to identify and analyze the problem. To correct any errors, you must exit from the Debugger and repeat the steps listed earlier. This section describes how to carry out these steps, both from the Programmer's Environment and from the system prompt.

To operate correctly, the Debugger must have access to the source modules and to the compiled p-code versions of your 4GL program. If these are not in your current directory, you should read "Specifying the Source Program Search Path" on page 8-19.

Working in the Programmer's Environment

This section includes procedures for creating, revising, compiling, and combining modules. It also describes how to execute a compiled program and invoke the Debugger. Before you can perform any of these tasks, however, you need to invoke the Programmer's Environment.

If your software has been installed according to the instructions in your installation letter, you can enter:

```
r4gl
```

at the system prompt to invoke the Programmer's Environment. A sign-on message is displayed, and after a pause, the **INFORMIX-4GL** menu appears. See the *INFORMIX-4GL Reference* for more information about the syntax of the **r4gl** command.

Creating a New Source Module

This section outlines the procedure for creating a new module. If your source module already exists but needs to be modified, you should skip ahead to the next section, "Revising an Existing Module."

To create a source module

1. Choose the **Module** option from the **INFORMIX-4GL** menu.
2. If you are creating a new **.4gl** source module, press `n` to select the **New** option of the **MODULE** menu.
3. Enter the name of the new module with the extension **.4gl**.

   The name must begin with a letter and can include letters, numbers, and underscores. The name must be unique among the files in the same directory and among the other program modules if it will be part of a multi-module program.

Revising an Existing Module

If you are revising an existing 4GL source file, rather than creating a new one, the procedures to begin an editing session are slightly different from the steps that were just described.

To revise a source module

1. Choose the **Module** option from the **INFORMIX-4GL** menu.
2. Select the **Modify** option of the **MODULE** menu.

   The screen displays the filenames of all of the **.4gl** source modules in the current directory and prompts you to select a source file to edit.
3. Use the arrow keys to highlight the name of a source module and press RETURN, or enter a filename.

   If you specified the name of an editor as the **DBEDIT** environment variable, an editing session with that editor begins automatically. If you did not specify a value for the **DBEDIT** environment variable, the screen prompts you to identify the text editor that you want to use.

4. Specify the name of a text editor, or press RETURN for **vi**, the default editor.

   Now you can begin an editing session by entering 4GL statements. See the *INFORMIX-4GL Reference* for more information on INFORMIX-4GL statements and programs.

5. When you have finished entering or editing your 4GL code, use an appropriate editor command to save your source file and end the text editing session.

Compiling a Source Module

The **.4gl** source file module that you create or modify is an ASCII file that must be compiled before it can be executed. After you save your file and exit from the editor, the screen prompts you to choose from among **Compile**, **Save-and-exit**, and **Discard-and-exit** options.

To compile a source module

1. Choose the **Compile** option to compile the module.

   After you choose **Compile**, the screen prompts you to select among the **Object**, **Runable**, and **Exit** options.

   The option you choose depends on whether your module is a complete program or whether it is one of several **.4gl** modules that together make up a complete program.

2. If the module is a complete 4GL program that requires no other modules, select **Runable**.

   This creates a compiled p-code version of your program module with the same filename but with extension **.4gi**.

3. If the module is one module of a multi-module 4GL program, select **Object**.

   This creates a compiled p-code version of your program module with the same filename but with the extension **.4go**. See also the procedures for combining program modules, which are described later in this section.

   If the compiler detects errors after either option, no compiled file is created, and the screen prompts you to select **Correct** or **Exit**. Follow the next two steps after an error.

4. Select **Correct** to resume the previous text editing session with the same text editor and **.4gl** source file but with error messages in the file.
5. Edit the file to correct the error and select **Compile** again.

   If an error message appears, repeat the previous steps until the module compiles without error.

   After the module compiles successfully, the screen prompts you again to select **Compile**, **Save-and-exit**, or **Discard-and-exit**.
6. Select the second option to save the compiled program. The **MODULE** menu appears again on your screen.
7. If your program requires *screen forms*, you must select **Exit** to return to the **INFORMIX-4GL** menu and then select **Form** to display the **FORM** menu.

   The Debugger does not display the source code of forms (files with the extension **.per**), but it can display the screen output that the compiled versions of these files produce.
8. If your program displays help messages, you must create a help file and compile it with the **mkmessage** utility.

See the *INFORMIX-4GL Reference* for information about designing and creating screen forms and about implementing help messages in 4GL programs.

Combining Program Modules

If the module that you compiled is the only module in your program, you are now ready to use the Debugger, and you can skip the steps that are described here. If your new or modified module is part of a multi-module 4GL program, however, you must combine all of the modules into a single program before you can use the Debugger.

To combine modules

1. If you are not at the **INFORMIX-4GL** menu, choose **Exit** until that menu appears.
2. Choose the **Program** option to display the **PROGRAM** menu.
3. If you are creating a new multi-module 4GL program, rather than modifying an existing one, choose the **New** option.

   The screen prompts you to enter the name of your program.

4. Enter the name that you want to assign to your program, without a file extension.

   The program name must begin with a letter and can include underscores (_) and numbers. After you enter a valid name, the PROGRAM screen appears with your program name in the first field, as shown in Figure 8-1.

Figure 8-1
NEW PROGRAM Screen

```
NEW PROGRAM: [ 4GL Globals Other Program_Runner Rename Exit
Edit the 4GL sources list.

                                                    Press CTRL-W for Help
 Program [           ]
 Runner [fglgo       ] Runner Path    [                                    ]
 Debugger[fgldb      ] Debugger Path [                                     ]

 4gl Source     4gl Source Path
 [           ] [                                          ]
 [           ]  [                                         ]
 [           ]  [                                         ]
 [           ]  [                                         ]
 [           ]  [                                         ]

 Global Source Global Source Path
 [           ]  [                                         ]
 [           ]  [                                         ]

 Other .4go     Other .4go Path
 [           ]  [                                         ]
 [           ]  [                                         ]
```

5. Press RETURN to select the **4GL** option.

   A message prompts you to enter the names of all the source modules of your program.

6. Enter the name of a module, without the **.4gl** file extension.

   Repeat this step for every module. If the module is not in the current directory or in a directory specified by the **DBSRC** environment variable, enter the pathname to the directory where the module resides.

7. If your program includes a **Globals** module, choose the **Globals** option and enter the corresponding information.

8. If your program includes any **.4go** modules that you have already compiled, choose the **Other** option to enter their filenames (and optionally, their pathnames).
9. After you have correctly listed all of the modules of your 4GL program on the NEW PROGRAM screen, choose the **Exit** option to return to the **PROGRAM** menu.
10. Choose the **Compile** option of the **PROGRAM** menu.

    This produces a file that combines all of your **.4gl** source files into an executable program. Its filename is the program name that you specified, with the extension **.4gi**. The screen lists the names of your **.4gl** source modules and displays the **PROGRAM** menu with the **Run** option highlighted.

Executing a Compiled Program

You could press RETURN to test for runtime errors by executing the compiled 4GL program. Because you have much greater control of program execution if you first invoke the Debugger, however, the usual procedure at this point is to begin a debugging session rather than to select the **Run** option.

Invoking the Debugger

The following procedure describes how to begin a debugging session.

To invoke the Debugger

1. If you are at the **INFORMIX-4GL** menu you must choose either the **Module** or the **Program** option before you can access the Debugger.
2. At either the **MODULE** or **PROGRAM** menu, press D to choose the **Debug** option.

   The screen prompts you to specify a 4GL program as your application for this debugging session. If any compiled files with the extension **.4gi** are in your current directory, their names appear below the prompt.

3. Select a program by using the arrow keys to highlight the name of a file, and then press RETURN.

   Alternatively, you can enter the name of a compiled 4GL program. You do not need to specify the **.4gi** extension. (You must supply the **.4go** extension to select a **.4go** file only if a file with the **.4gi** extension has the same filename.)

Working at the Command Line

The same **.4gl** source files and compiled **.4go** and **.4gi** p-code files can also be created at the operating system prompt. Figure 8-2 shows the process of creating, compiling, and running or debugging a single-module program from the command line. Here the rectangles represent specific operating system commands, and the circles represent disk files. Arrows indicate whether a file serves as input or output for a process.

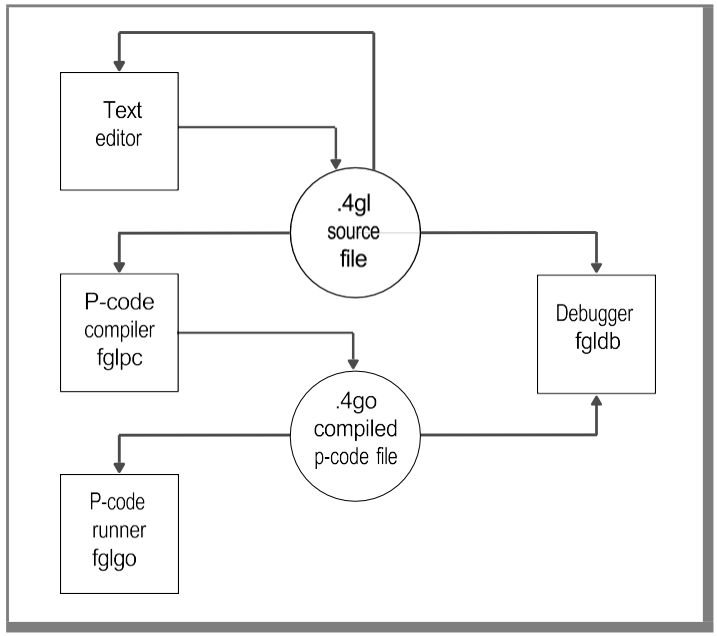


Figure 8-2
Debugging a Single-Module Program

This diagram is simplified and ignores the similar processes by which forms and other components of 4GL applications are compiled and executed:

- The cycle begins in the upper-left corner with a text editor, such as **vi**, which you use to produce a 4GL source module.
- You can then compile the program module by using the **fglpc** p-code compiler. (If error messages are produced by the compiler, you must locate the errors in the **.err** file and edit the **.4gl** file to correct them. Then recompile the corrected **.4gl** file.)
- Next, you can invoke the Debugger at the system prompt by entering the command:

  ```
  fgldb filename
  ```

  where filename specifies a compiled source file.
- You might need to modify the source file to eliminate runtime errors that the Debugger identifies. You can then recompile and retest the 4GL program. When it is ready for use by others, they can use the **fglgo** program to execute the compiled program.

The correspondence between commands and Programmer's Environment menu options is summarized by the list that follows.

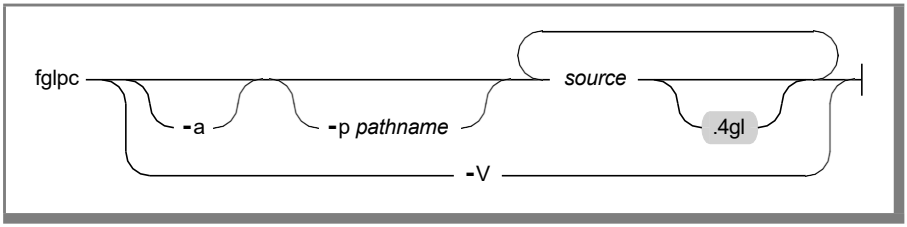
Command	Description	Menu Options
vi	UNIX system editor	New, **Modify**
fglpc	4GL P-Code Compiler	Compile
fglgo	4GL P-Code Runner	Run
fgldb	4GL Interactive Debugger	Debug

Creating or Modifying a 4GL Source File

Use your system editor or another text-editing program to create a **.4gl** source file, or to modify an existing file. Refer to the documentation for your editor and to *INFORMIX-4GL Concepts and Use* for details.

Compiling a Source File

You cannot use the Debugger to examine a 4GL program until you have compiled each source module into a **.4go** file. You can compile source modules from the system prompt using the **fglpc** command as follows.



Element	Description
fglpc	The required name of the command file.
-V	Displays the version number of the software.
-a	Causes your compiled program to check array bounds at runtime.
-p *pathname*	Stores object (**.4go**) and error (**.err**) files in directory pathname.
***source*.4gl**	The name of your 4GL source module. You do not need to specify the **.4gl** extension. You can specify any number of source files.

Unless you specify the **-V** option, this command creates a compiled version of each **.4gl** source module. Each compiled module has the same filename as the corresponding source file but with the extension **.4go**.

If you specify the **-V** option, the screen displays the version number of your SQL and p-code compiler software. Any other command options are ignored. After displaying this information, the program terminates without invoking the p-code compiler.

If you specify a nondefault directory with the **-p** *pathname* option, the **.4go** and **.err** files are stored in directory *dir*. Unless you specify the **-p** *pathname* option, compiled modules are stored in the current directory.

Because the **-a** option requires additional processing, you might want to use this option only during development for debugging purposes.

If an error occurs during compilation, a message is displayed, and a file ***source*.err** is created. You can look in ***source*.err** to find where in your code the error occurred. The ***source*.err** file is created in the current directory or in the directory that you specify with the **-p** *pathname* option.

You can specify any number of source files, in any order. You do not need to specify their **.4gl** file extensions.

Examples

The following command compiles a 4GL source file, **single.4gl**, and creates a file called **single.4go** in the current directory:

```
fglpc single.4gl
```

The next command line compiles two 4GL source files:

```
fglpc -p /u/ken fileone filetwo
```

This generates two compiled files, **fileone.4go** and **filetwo.4go**, and stores the compiled files in subdirectory **/u/ken**. Any compiler error messages are saved in files **fileone.err** or **filetwo.err** in the same directory.

Concatenating Multi-Module Programs

If a program has several modules, the compiled modules must all be concatenated into a single file with a **.4gi** or **.4go** file extension. For example:

```
cat  file1.4go  file2.4go  ...  fileN.4go  >  new.4gi
```

combines the list of **.4go** files into a file called **new.4gi**. (This step is represented in Figure 8-3.)

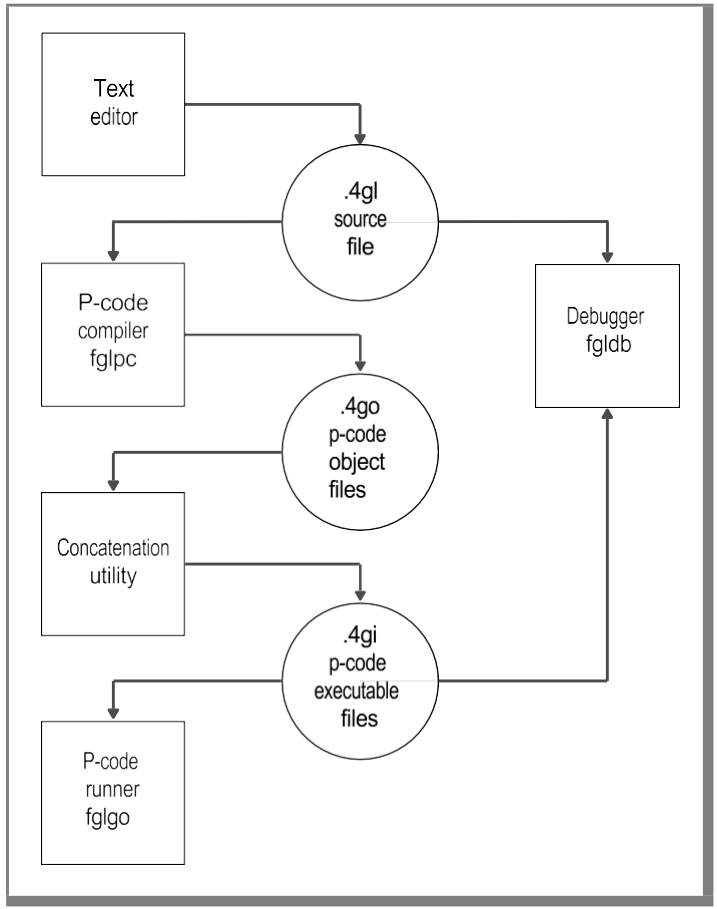


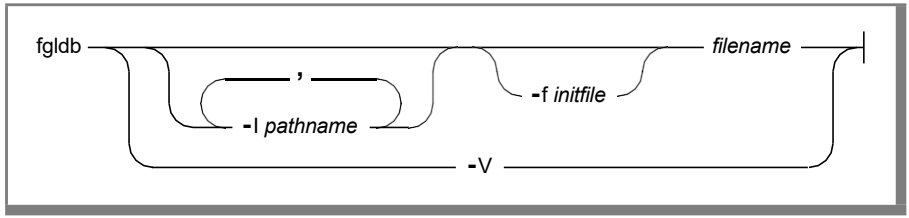
Figure 8-3 *Debugging a Multi-Module Program*

Throughout this manual, the **.4gi** extension is used to designate runnable programs that have been compiled and concatenated. You might want to follow this convention in naming files because only **.4gi** files are displayed from within the Programmer's Environment. This is also a convenient way to distinguish complete program files from individual modules of a multi-module program.

If your 4GL program calls C functions or INFORMIX-ESQL/C functions, you must also follow the procedures that are described in Appendix B, "Calling C Functions," before you can use the Debugger.

Invoking the Debugger

After you have compiled your 4GL program, you can invoke the Debugger by entering the **fgldb** command at the system prompt.



Element	Description
fgldb	The name of the Debugger command file.
-V	Displays the version number of the software.
-I	Specifies a nondefault search path for 4GL source files.
-p *pathname*	An optional search path specification.
-f	An optional symbols to specify a nondefault initialization file.
initfile	The name of an optional **.4db** initialization file.
filename	The name of a compiled 4GL program.

Unless you select the **-V** option, a **fgldb** command loads the Debugger and specifies *filename* as the 4GL application program. An error message appears unless the Debugger can access all of the files specified in the command line, and any corresponding **.4gl** source files. If *filename* is outside your current directory, you must prefix it with a pathname.

If you do not specify a file extension, the Debugger looks for a file called ***filename*.4gi**. If it cannot locate this file, it searches for a file called ***filename*.4go**.

Once you specify a valid program, you cannot select a different program as your current application unless you exit from the Debugger.

If you specify the **-V** option, the screen displays the version number of your SQL and Debugger software. After displaying this information, the program terminates without beginning a debugging session, and the system prompt returns.

The **-I** *pathname* specification is optional. Use it if any program modules are not in your current directory search path. The next section of this chapter describes other ways to specify the directories in which the Debugger searches for 4GL source files.

Use a comma between pathnames if you specify several directories. Blank characters are not allowed in a list of pathnames.

The **-f** *initfile* specification is also optional. The initialization file must have the extension **.4db**, but you do not need to include this extension in the command line.

See "Establishing Parameters from Files" on page 8-45 for a description of how **.4db** files can be used to customize your debugging environment.

The filename of a **fgldb** command line cannot be a 4GL program that calls C functions. Appendix B describes the special procedures for using the Debugger to analyze 4GL programs that call C functions or ESQL/C functions.

Examples

Enter the following command to use the Debugger to examine the compiled 4GL program file called **cust_order.4gi**:

```
fgldb cust_order
```

The next command:

```
fgldb -f template customer.4go
```

specifies **customer.4go** as the compiled program and begins the debugging session by reading the Debugger commands in a file called **template.4db**.

The following command:

```
fgldb  -I  /u/myfiles,/g/testfiles  catalog.4go
```

specifies that the source file search path begins with the directories **/u/myfiles** and **/g/testfiles** and that the program **catalog.4go** is the current program. If you do not specify the extension **.4go**, the Debugger makes **catalog.4gi** the current 4GL program if it finds a compiled program with that name in the current directory.

The following command:

```
fgldb  -I  /u/myfiles  /g/testfiles/catalog.4go
```

specifies that the Debugger will begin searching for source files in directory **/u/myfiles** and that the program **catalog.4go** in directory **/g/testfiles** is the current program.

The following command:

```
fgldb  -V
```

displays the release versions of your SQL software and of your p-code compiler and then returns control to the operating system. This option might be helpful in some troubleshooting situations.

Specifying the Source Program Search Path

The Debugger supports several methods by which you can specify the names of the directories that contain your 4GL source files. The following list shows the order in which directories are searched. If two source files have the same name, the Debugger uses the file whose directory was specified by a method nearest the top of this list:

1. Your current directory. The Debugger always searches for files in the directory from which you invoked the Debugger.
2. The directory associated with the 4GL program. If you include a pathname in the specification of the 4GL application program, the Debugger also searches that directory for source modules.

3. The directories that you can specify after the **-I** symbols if you invoke the Debugger from the system prompt.
4. The directories specified by the **DBSRC** environment variable. See Appendix A, "Environment Variables," for information on how this is specified.

This is the directory search order at the beginning of a debugging session. Directories can also be specified by the USE command. The USE command allows you to specify a list of directories to be searched during a debugging session before any of the directories listed previously. An option of the USE command enables you to specify a search path that *replaces* your current source file search path. See Chapter 9, "The Debugger Commands," for the syntax of the USE command.

The pathnames and search order of directories that can be specified by these methods are only in effect during the current debugging session. Directories that hold databases required by the 4GL application program must be specified by the **DBPATH** environment variable. See Appendix A for a description of the **DBPATH** and **DBSRC** environment variables.

Specifying Keyboard Aliases

The Debugger assigns default command strings to the first nine function keys of many terminals unless you have redefined those keys in an initialization file. ("Establishing Parameters from Files" on page 8-45 describes initialization files.) The default command strings assigned to these keys were listed in Chapter 1, "Introduction to the Debugger," but are reproduced here:

```
alias f1 = help
alias f2 = step
alias f3 = step into
alias f4 = continue
alias f5 = run
alias f6 = list break trace
alias f7 = list
alias f8 = dump
alias f9 = exit
```

You can use the ALIAS command to replace these command strings with other strings, or to assign Debugger command strings to any alphanumeric key or sequence of keys that starts with a letter. This is a convenient feature because it allows you to assign command strings to any key or sequence of keystrokes that you specify.

If you enter a Debugger command line that includes one or more aliases, the Debugger screen echoes your keystrokes and then displays the expanded form of your command, substituting for the aliases. Aliases can contain other aliases. You cannot alias Debugger commands such as **Interrupt**, **Redraw**, **Screen**, or **Toggle**, which are invoked by control characters rather than by keywords.

If you program your function keys by some other method, you cannot use their default aliases.

Any keyboard assignments made by the 4GL application program that you are debugging override these aliases when the program requires input. The standard meanings of the keys are restored after you exit from the Debugger. Chapter 9 describes the syntax of the ALIAS command in detail.

The Debugger Screens and Windows

The Debugger allows you to monitor a debugging session through the following screens and windows:

- **Debugger screen.** This screen consists of a Source window and a Command window.

  The Command window accepts and displays the 50 most recent lines of your Debugger commands, their output, and any error messages. It occupies the lower part of your Debugger screen display.

  The Source window displays the name, line numbers, and source code of the current 4GL module. This appears in the upper part of your Debugger screen display.

- **Application screen.** This screen displays screen output from your 4GL program and overwrites the Debugger screen. You can use the APPLICATION DEVICE command to direct this full-screen display to a second video terminal.

Besides these displays of the Debugger screen and the Application screen, the Debugger supports two additional displays:

- **Help screen.** This full-screen display shows help menus and help messages that provide information about the Debugger. You can only access the Help screen from the Command window.
- **Operating system display.** If you type an exclamation point (!) followed by an operating system command, control returns temporarily to the operating system, and you see the display (if any) of the command.

Descriptions of the Debugger Displays

The sections that follow describe each of the Debugger displays. Although you can sometimes see several of them simultaneously, at a given moment you can enter information at only one of them. You can select this active display (also called your *current* window or screen) by using appropriate commands or control keys. Figure 8-4 on page 8-25 summarizes how to switch your current display.

Command Window

Occupying the lower portion of the Debugger screen, the Command window becomes your current window when you successfully invoke the Debugger and specify your current 4GL program. In the Command window, you can perform the following tasks:

- Enter Debugger commands.
- Scroll and search the command buffer.
- Save or redraw the Debugger screen.
- Observe Debugger output and error messages.

You can switch from the Command window to any other display with the following commands:

- VIEW switches to the Source window.
- CTRL-T switches to the Application screen.

- The HELP command switches to the Help screen.
- The exclamation point (!), followed by a system command, transfers control temporarily to the operating system.

From the Source window, pressing the **Interrupt** key (typically CTRL-C) switches to the Command window. If the EXITSOURCE parameter is ON, typing any alphabetic character switches to the Command window and echoes the character after the $ prompt.

From the Application screen, you can return to the Command window by pressing the **Interrupt** key. If you have toggled to display the Application screen, you can return to the Command window by pressing CTRL-T or any key except CTRL-S, CTRL-P, CTRL-Q, or CTRL-R.

From a Debugger Help screen, you can press RETURN until the Command window reappears.

From the operating system display, you can return to the Command window by pressing any key.

Source Window

If you are working in the Command window, the VIEW command makes the Source window your current window. This window displays and highlights the name, module, and source code of the current 4GL function. In the Source window, you can perform the following tasks:

- Scroll and search the source module.
- Display the 4GL statement at which execution stopped.
- Save or redraw the Debugger screen.
- Execute an operating system command.

Besides switching to the operating system display with the exclamation point, you can switch from the Source window to other displays:

- The **Interrupt** key switches to the Command window.
- CTRL-T displays the Application screen.

If you have used CTRL-T to switch to the Application screen from the Source window, and EXITSOURCE is OFF, pressing any key (except the control keys CTRL-S, CTRL-Q, CTRL-P, or CTRL-R or the **Interrupt** key) returns you to the Source window. If EXITSOURCE is ON, any alphabetic key selects the Command window.

Application Screen

This screen appears when the current 4GL application requires keyboard input or produces screen output.

The Application screen appears if you press CTRL-T from the Debugger screen, or if the 4GL program produces output when AUTOTOGGLE is ON. When the AUTOTOGGLE parameter is OFF, the Application screen is not displayed automatically when program output is produced, but it appears when keyboard input is required. When the 4GL program requires input, you can either supply the requested input or switch to the Command window by pressing the **Interrupt** key.

If the 4GL program is not requesting input, pressing any key (except CTRL-S, CTRL-Q, CTRL-P, CTRL-R, or the **Interrupt** key) restores your previous window.

The Application screen is empty when you press CTRL-T unless the 4GL program has produced screen output. You must first issue a RUN or CALL command from the Command window before the program or a function can produce any output. The CLEANUP command clears the Application screen.

Help Screen

Enter `HELP` (or `H`) at the Command window to make the Help screen your current window.

Unless you specify a Debugger command after the HELP keyword, the Help screen initially displays a list of command names. If you select a topic from this list, the Help screen then displays the first page of a message about the topic. If you enter `help all` at the Command window, the Help screen displays a synopsis of every Debugger command.

After you have read the first page of a message, type `S` to display the next page, or else type `R` to return to the Command window.

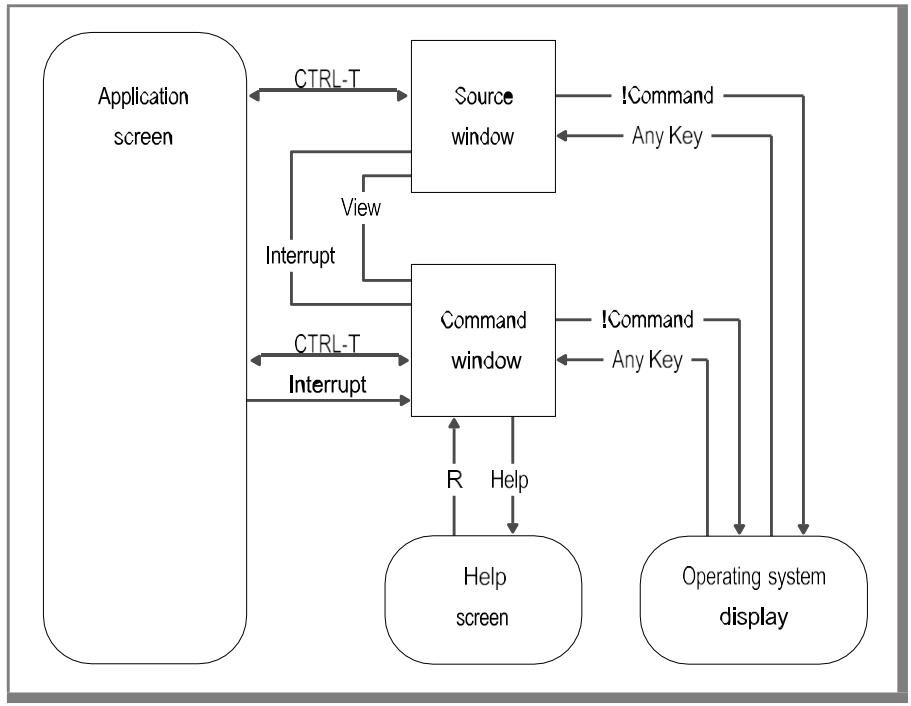
Operating System Display

To execute an operating system command, type an exclamation point (!) at the Command or Source window, and enter the command or the name of an executable command file. If this produces screen output, the Command and Source windows scroll up to make room for your output. The command can invoke an interactive program that requires keyboard input.

After the command terminates, the Debugger prompts you to press any key, which returns you to the Command or Source window. If you again type an exclamation point, you can enter another command line without returning to the Debugger screen.

Figure 8-4 depicts the Debugger windows and screens and indicates some of the keyboard commands to switch your current window from one to another.

Figure 8-4
A Map of the Debugger Windows and Screens



Setting Terminal Display Parameters

The Debugger supports several parameters that allow you to control the Source and Command windows and the interaction of the Debugger screen with the Application screen.

The functions of these parameters and the commands to change their values are described in the sections that follow. When you invoke the Debugger, they have the default values listed in the following table unless other values are specified by a **.4db** initialization file (as described later in this chapter).

Parameter	Command	Default
AUTOTOGGLE	TURN ON/OFF	ON
DISPLAYSTOPS	TURN ON/OFF	ON
EXITSOURCE	TURN ON/OFF	ON
PRINTDELAY	TURN ON/OFF	OFF
SOURCETRACE	TURN ON/OFF	OFF
TIMEDELAY SOURCE	TIMEDELAY	1
TIMEDELAY COMMAND	TIMEDELAY	0
SOURCE LINES	GROW	9 *
COMMAND LINES	GROW	10 *
APPLICATION DEVICE	APPLICATION DEVICE	(no second terminal)

** Default window size for a standard 24-line terminal*

If you exit to the Programmer's Environment after changing any of these parameters and subsequently return to the Debugger to work with the same 4GL program, your modified values are in effect. If you exit to the operating system or select a different 4GL program, however, default values replace your modified values when you begin another debugging session.

Parameters Controlled by the TURN Command

You can use the TURN command in the Command window to alter the values of five of the terminal display parameters. The function of each is described in the sections that follow.

AUTOTOGGLE

When the AUTOTOGGLE display parameter is ON, the Debugger displays the Application screen whenever your program requests input from the user or generates output. If AUTOTOGGLE is turned OFF, the Debugger switches to the Application screen only when the 4GL program requires input. To view the Application screen at any other time, you must press CTRL-T from the Command or Source window to toggle the display.

If you are running a program that produces frequent output, it might be easier to work in the Command or Source window when this parameter is turned off. The default value for AUTOTOGGLE is ON.

DISPLAYSTOPS

When DISPLAYSTOPS is ON, the Debugger changes the display in the Source window to show the 4GL statement that is currently executing and highlights the next statement to be executed when execution stops. If DISPLAYSTOPS is OFF, the contents of the Source window are not modified, and when execution stops, the Debugger displays the next statement to be executed in the Command window. You might invoke a TURN OFF DISPLAYSTOPS command if you have lengthy comments or variable definitions in your program that you want to remain in the Source window for your reference during the session. The default for DISPLAYSTOPS is ON.

EXITSOURCE

When EXITSOURCE is ON and the cursor is in the Source window, pressing any alphabetic key or the **Interrupt** key makes the Command window the current window. The Debugger then echoes and executes any command that you type at the Source window.

If both AUTOTOGGLE and EXITSOURCE are ON, for example, you can type RUN when the Source window is your current window. The result is that control switches to the Command window, where RUN is entered. The Application screen replaces the Command window as the current window if the program prompts for keyboard input or displays output. When EXITSOURCE is OFF, only the **Interrupt** key switches you from the Source window to the Command window. The default value for EXITSOURCE is ON.

PRINTDELAY

When PRINTDELAY is ON, the screen is updated in multiple-line blocks when a Debugger command (such as LIST) sends output to the Command window. When PRINTDELAY is OFF, the Command window is updated a single line at a time, and multiple-line output from Debugger commands scrolls up the Command window. When you are tracing execution, OFF is usually better. On some terminals, or if your system is slow, a TURN ON PRINTDELAY command results in a more timely display. You might want to experiment with both values to find your preferred setting. The default value for PRINTDELAY is OFF.

SOURCETRACE

If SOURCETRACE is ON, the Debugger highlights each line of code as it executes, and modifies the contents of the Source window as necessary. When SOURCETRACE is OFF, the Debugger does not highlight each line of code as it executes. One effect of a TURN ON SOURCETRACE command is to increase the time required for a debugging session. It can be very useful, however, when you are first becoming familiar with a program or when you are working with a relatively small program. Alternatively, it can be turned on for sections of a program that require particular attention and turned off otherwise. The default value for SOURCETRACE is OFF.

This concludes the list of display parameters that are controlled by the TURN command.

Parameters Controlled by the TIMEDELAY Command

Two terminal display parameters, TIMEDELAY SOURCE and TIMEDELAY COMMAND, are controlled by the TIMEDELAY command. The TIMEDELAY command resembles the 4GL SLEEP statement. The integer argument of a TIMEDELAY command line specifies the number of seconds that elapse before the display is updated. This determines how quickly the display changes in the Source window or in the Command window.

TIMEDELAY SOURCE

The TIMEDELAY SOURCE parameter specifies the number of seconds that elapse before the next 4GL statement is highlighted in the Source window when the SOURCETRACE parameter is ON. The larger the value, the longer the delay before the next statement is highlighted in the Source window. The initial default value of TIMEDELAY SOURCE is `1`.

If you would like to highlight each line of source code for a shorter period, you could enter the following TIMEDELAY command at the Command window prompt:

```
timedelay source 0
```

This is equivalent to the command:

```
timedelay 0
```

because SOURCE is the default window specification of the TIMEDELAY command. This example selects a delay of zero seconds while each line of source code is highlighted. This could reduce the time required for a debugging session.

Because values of the DISPLAYSTOPS, SOURCETRACE, and TIMEDELAY SOURCE parameters all affect the display of both the Application and the Source windows, they do not work independently. The TIMEDELAY SOURCE parameter has no effect unless SOURCETRACE is ON.

TIMEDELAY COMMAND

The TIMEDELAY COMMAND parameter determines the speed at which each new line of Debugger output is displayed in the Command window. This controls how quickly commands such as LIST and READ can send their next line of output to the Command window. The initial default value is zero, meaning that no delays are inserted.

Sometimes, however, you might want a slower display. A READ command, for example, can execute a list of Debugger commands that produce output in the Command window. These might scroll above the top of the Command window faster than you can read them or before you can press CTRL-S to stop the display. In situations like this, resetting the TIMEDELAY COMMAND parameter to a nondefault value can allow you to read the Command window display more easily. For example, the command:

```
timedelay  command  1
```

waits for one second before sending another line of output (or an error message) to the command buffer. The integer argument of a TIMEDELAY COMMAND command specifies the number of seconds to delay before successive lines of output appear in the Command window.

Parameters Controlled by the GROW Command

The relative size of the Command and Source windows is specified by two terminal display parameters, SOURCE LINES and COMMAND LINES, which are both controlled by the GROW command. The syntax and options of GROW resemble those of TIMEDELAY, except that the numeric argument of GROW can sometimes have negative values.

If your terminal can display *L* lines, the sum of COMMAND LINES and SOURCE LINES values is always *(L*-5*)*. The minimum size of either window is one line. The maximum value is 18 lines on a standard (24-line) terminal.

SOURCE LINES

The SOURCE LINES parameter specifies the number of lines of source code that can be displayed in the Source window without scrolling. On a 24-line screen, it has a default value of `9`. This value can be changed by using the GROW command. For example, the command:

```
grow source 3
```

adds 3 to the current value of SOURCE LINES, expanding the Source window by three additional lines. The command:

```
grow 3
```

has the same effect because SOURCE is the default window specification of a GROW command.

Similarly, the command:

```
grow -2
```

reduces the size of the Source window by two lines because the numerical argument of the GROW command, here **-2**, is added to the current value of SOURCE LINES.

COMMAND LINES

The number of lines visible in the Command window is specified by the COMMAND LINES parameter. This has a default value of `10` on a standard (24-line) terminal. The value can be changed by using the GROW command. For example, the command:

```
grow command 4
```

increases the size of the Command window by four additional lines. The command

```
grow -4
```

has the same effect because SOURCE is the default window specification, and reducing SOURCE LINES by 4 increases COMMAND LINES by the same amount.

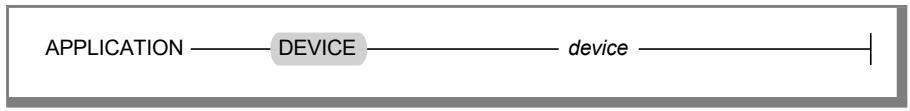
The APPLICATION DEVICE Command

Besides the display parameters that have already been described in this chapter, the Debugger allows you to specify an *application device*. This allows you to redirect screen output of the current 4GL program to another video terminal, rather than to the terminal from which you invoked the Debugger. The default is to direct 4GL application output to the same terminal that invoked the Debugger.

The APPLICATION DEVICE command is a Debugger feature that can be helpful if you have access to two identical video display terminals on a multiuser system, or two terminals that can support the same **termcap** or **terminfo** entries. The APPLICATION DEVICE command dedicates the second physical terminal to the Application screen. This allows you to monitor continuously the output display from the application program that you are analyzing, regardless of which Debugger display is your current window.

Even if you use this command to redirect output to the screen of a second terminal, you cannot use the keyboard of the second terminal to enter input. Any input to the application program must be entered at the keyboard of the terminal from which you invoked the Debugger.

To specify a second terminal for the Application screen, you must enter a command of the following form.



For example, the command:

```
application  device  /dev/ttyp14
```

designates as the application device the terminal whose device name is **/dev/ttyp14**. Both must be logged in by the same account name. Both use the **termcap** or **terminfo** entry for the terminal from which you invoked the Debugger.

To restore the default value of this parameter (no separate application device), you must use the EXIT command to terminate the Debugger and return to the operating system.

Establishing Breakpoints and Tracepoints

Among the most powerful features of the Debugger are commands to specify breakpoints and tracepoints. They can control when the 4GL program stops or what diagnostic information the Debugger displays as you run your program. They help you to monitor changes in 4GL variables and analyze program logic. You can also specify commands for the Debugger to execute when a breakpoint or tracepoint is reached.

The BREAK Command

The BREAK command creates *breakpoints*. These can suspend program execution at a 4GL statement or function when a variable changes or if logical conditions are satisfied. The complete syntax of BREAK is described in Chapter 9, "The Debugger Commands."

The following examples illustrate different criteria that you can establish for a breakpoint to take effect:

- **Break at a specific line**. If you enter:

  ```
  break 30
  ```

  execution stops when the Debugger reaches line 30. If line 30 is part of a multiple-line statement, execution stops at the first line of the statement. If line 30 is not executable, it stops at the next executable statement or at the last line of the function, whichever comes first. An error message appears if the module contains no executable statements, or fewer than 30 lines. If you enter:

  ```
  break -6 30
  ```

  execution stops the sixth time that it reaches line 30. If you enter:

  ```
  break 30 if x > 5
  ```

  execution stops when it reaches line 30 of the current module if the value of variable **x** is greater than 5.

  ```
  break fast.30 if x > 5 { print x }
  ```

  Execution stops at line 30 of module **fast.4gl** if variable **x** is greater than 5. After this breakpoint suspends execution, the Command window shows the value of variable **x**.

- **Break at a specific function**. If you enter:

```
break funca
```

execution stops if the function called **funca** is entered. If you enter:

```
break 'alfa' funca if i = 7
```

execution stops if the function called **funca** is entered when the value of variable **i** is equal to `7`. Debugger commands such as DISABLE, ENABLE, NOBREAK, and WRITE can reference this named breakpoint as *alfa*.

(See Chapter 9 for more information on these commands.)

- **Break when a variable changes.** If you enter:

```
break vara
```

execution stops if the value of the variable **vara** changes. If you enter:

```
break vara { print vara >> filea; dump all }
```

execution stops if the value of **vara** changes. Then the name and value of **vara** are saved in a file called **filea**, and a DUMP ALL command displays the values of the variables in the current function, as well as the values of all global variables. If you enter:

```
break vara if status = 0
```

execution stops if the value of **vara** changes while the value of the STATUS flag is zero.

- **Break if a logical condition is TRUE.** If you enter:

```
break if status = 100
```

execution stops if the value of the **status** flag is `100`. If you enter:

```
break if impact = 1 AND n > 9 { let n = 0; continue }
```

execution stops if the value of the variable **impact** is `1` and **n** is greater than 9. After execution is suspended, the value of **n** is reset at zero. The Debugger then executes a CONTINUE command, which immediately resumes program execution. If you enter:

```
break (dropship) if impact
```

execution stops if local variable **impact** that was defined in function **dropship** is assigned a nonzero value.

Interactions Among Breakpoints Set on Variables

As these examples show, the name of a program variable can be specified in a BREAK command. The breakpoint stops execution if the variable changes, or if an expression that includes the variable becomes TRUE. If there is a change in the value of *any* variable associated with an active breakpoint, the Debugger checks *all* active breakpoints that reference variables.

This can result in interactions among breakpoints, if several BREAK commands reference variables. Suppose, for example, that two of your currently active breakpoints have been specified by the following commands:

```
break if x
  . . .
break if i = 100
```

The first suspends program execution if variable **x** is assigned a nonzero value. The second stops the program when variable **i** becomes equal to 100.

In this example, the Debugger suspends program execution when any of the following changes occur in the values of **i** or **x**:

- **x** becomes TRUE (that is, unequal to zero)
- **i** becomes equal to 100
- **x** changes when **i** is equal to 100
- **i** changes when **x** is TRUE.

When either of the first two events listed previously occurs, stopping is the precise effect that you requested when you specified one of these breakpoints. If you resume execution with a CONTINUE command, however, you might not have intended the program to stop after either of the last two events occurs.

To avoid this problem, you could enter a DISABLE command after the first breakpoint takes effect. You can do this automatically by substituting a command such as:

```
break "truex" if x { disable "truex" }
```

for the first BREAK command in this example. This creates a self-disabling breakpoint that cannot stop the program more than once unless you reset it with an ENABLE command.

When you have many active breakpoints, it might help to specify an identifying PRINT message in each BREAK command. In this way, you can determine which breakpoint caused each break. For example, a named breakpoint that identifies itself could be set on variable **y** by the following command:

```
break "eg" if y { disable "eg"; print "eg" }
```

Resuming Execution After a Breakpoint

When execution is suspended by a breakpoint, you can use a PRINT, DUMP, or LET command to evaluate or reassign program variables in any 4GL functions that have not yet returned, or any global variables. After you have observed the program and invoked whatever Debugger commands suit your purpose, you have several options for resuming execution:

- You can use a CONTINUE command to resume execution at the breakpoint for an indefinite number of statements.
- You can use a STEP command to resume execution at the breakpoint for a specific number of 4GL statements. The NOBREAK option of STEP ignores any breakpoints that are encountered while STEP executes those statements.
- You can use a CALL command to execute any function, including **main**. (Before a CALL command, you might need to issue a CLEANUP command to reinitialize the program.)
- You can use a RUN command to reinitialize and restart the program from the beginning.

Removing or Disabling a Breakpoint

A breakpoint can be deleted by a NOBREAK command, or deactivated by a DISABLE command. An inactive breakpoint can be reactivated by an ENABLE command.

The TRACE Command

The command establishes *tracepoints*. These cause the Debugger to display information about the current 4GL program when a statement or function is reached or when a variable changes. The complete syntax of TRACE is described in Chapter 9. The following examples illustrate features of a 4GL program that you can monitor with a tracepoint:

- **Trace a specific line.** When you enter:

  ```
  trace 40
  ```

  if execution reaches line 40 of the current module, the Debugger appends the statement in line 40 to the command buffer. If line 40 is part of a multiple-line statement, it appends the entire statement. If line 40 is not executable, it appends the next executable statement or the last statement of the function, whichever comes first. An error message appears if the module contains no executable statements or fewer than 40 lines. When you enter

  ```
  trace fast.40 { print x }
  ```

  if the next statement after line 39 of the module called **fast.4gl** is executed, the statement is added to the command buffer, and a PRINT command evaluates variable **x**.

- **Trace a specific Variable.** When you enter:

  ```
  trace varb
  ```

  a message appears in the command buffer if the value of the variable called varb changes. The message displays the qualified name and new value of the variable, and the function, line number, and module that assigned the new value. When you enter:

  ```
  trace (funcb) varb {print varb >> fileb; dump}
  ```

  if local variable **varb** in function **funcb** changes, the message in the previous example appears in the command buffer. Then a PRINT command saves in file **fileb** the name and the new value of **varb**. Then a DUMP command appends output to the command buffer, evaluating the variables in the current function.

- **Trace a specific function.** If you enter:

```
trace funcb
```

  a message appears in the command buffer whenever the function funcb is entered, showing its name, the values of any arguments, and the line number and function from which it was called. Another message appears whenever **funcb** returns, showing any returned values. When you enter:

```
trace funcb >> funcb.out
```

  if function **funcb** is entered or returns, the messages in the previous example are saved in a file called **funcb.out**, not in the command buffer. If that file does not exist when the tracepoint is reached, the Debugger creates it. When you enter:

```
trace funcb { print i >> i.out } >> funcbtr
```

  if the function **funcb** is entered or returns, the Debugger saves the same messages in a file called **funcbtr** in the current directory. Whenever the funcb function is entered, a PRINT command saves the name and current value of variable **i** in a file called **i.out**. (When you trace a function, any embedded commands are executed when the function is entered rather than when it returns.)

- **Trace all functions.** If you enter:

```
trace functions
```

  the Debugger adds a message to the command buffer whenever any function is entered, showing the name of the function, any arguments, and the line number and function of the statement that called it. When any function returns, its name and returned values appear in the command buffer. If you enter:

```
trace 'beta' functions >> funcs.out
```

  whenever any function is entered or returns, the Debugger saves the messages described in the previous example in a disk file called **funcs.out**. Other Debugger commands such as DISABLE, ENABLE, NOTRACE, and WRITE can refer to this named tracepoint as **beta**. (See Chapter 9 for more information on these commands.)

Unless your system or terminal is relatively slow, it is difficult to read tracepoint output in the Command window if your 4GL program prompts for input because the Debugger automatically switches to the Application screen. If you want more time to read tracepoint output when it appears in the Command window, you can use a TIMEDELAY COMMAND command. Alternatively, you can use the **Interrupt** key to switch to the Command window.

A tracepoint can be deleted by a NOTRACE command or deactivated by a DISABLE command. An inactive tracepoint can be reactivated by an ENABLE command.

The Debugger does not impose any limit on the number of breakpoints and tracepoints that you can specify, apart from the memory or mass storage capacity of your system.

Restrictions on BREAK and TRACE Commands

The previous examples of BREAK and TRACE commands apply to 4GL functions that contain executable statements and to variables and to line numbers in 4GL functions. You can also establish a breakpoint or tracepoint at the line number of a 4GL statement that calls a C function or an ESQL/C function.

An error message results if a BREAK or TRACE command specifies a line number or the name of a variable inside a C function or a 4GL library function, or the name of a variable that exists only in a C-language module. A TRACE command (but not a BREAK command) can specify the name of a C function. Appendix B describes using the Debugger with 4GL programs that call C or ESQL/C functions.

The optional *commands* that you can specify in braces ({ }) in a TRACE command cannot include program execution commands such as CALL, CONTINUE, RUN, or STEP because the program does not stop executing when you reach a tracepoint.

The TRACE command can specify any other commands except those that require a control character (**Interrupt**, **Screen**, **Redraw**, and **Toggle**). BREAK commands can specify any Debugger commands except these nonkeyword commands. Use a semicolon to separate consecutive commands.

A breakpoint or tracepoint results in an error message, however, if any variable or function specified in the *commands* list of a BREAK or TRACE command violates the rules in "Active Functions and Variables" on page 9-20. Other error messages appear if the *commands* list contains other syntax errors.

If you assign a name to a breakpoint or tracepoint, the name must be unique and must begin with a letter. Regardless of whether you assign a name, the Debugger assigns a unique reference number to every breakpoint and tracepoint. This number is displayed in the Command window after a valid BREAK or TRACE command.

Displaying and Copying Parameters

Earlier sections of this chapter described commands to establish or modify the source file search path, keyboard aliases, terminal display parameters, breakpoints, and tracepoints. The Debugger command language supports other facilities related to the user interface of the Debugger. This section describes commands that you can use with debugging environment parameters to accomplish the following tasks:

- List environment values on the screen.
- Save environment values in a disk file.
- Replace environment values with values from a disk file.

Displaying Values with the LIST Command

To examine the current values of the terminal display parameters, enter the following command at the command prompt:

```
list display
```

The following output from a LIST DISPLAY command describes the default values of terminal display parameters:

```
TERMINAL  DISPLAY  STATE
autotoggle            on
displaystops          on
sourcetrace           off
exitsource            on
printdelay            off
timedelay source      1
timedelay  command    0
source  lines         9
command  lines        10
```

If you had specified a separate application device, the name of that parameter and its terminal device name would also have appeared in the LIST DISPLAY output.

To see your current breakpoints, enter:

```
list  break
```

The Command window displays the reference numbers, optional names, and complete specifications of all your current breakpoints, if any exist. Active and inactive breakpoints are listed separately, in the order of their reference numbers. The LIST TRACE command displays the corresponding information about all your current tracepoints in the same format as LIST BREAK.

Specifying both options (LIST BREAK TRACE) displays all your current breakpoints and tracepoints. The LIST command with no option displays all your current display parameters, breakpoints, and tracepoints.

Displaying Values with ALIAS

The ALIAS command can also display information about your current debugging environment. If you enter the command:

```
alias  *
```

at the Command window, the screen displays all of the current aliases that you have assigned to function keys or to command strings. The initial default aliases are listed in the next section, "Establishing Parameters from Files."

Suppose that you established three additional aliases by means of the commands:

```
alias y1 = disable all
alias y2 = grow -1
alias y3 = { y1;y2 }
```

Then the command:

```
alias *
```

would list all your aliases in alphabetical order, beginning with the default aliases, and then the following aliases:

```
y1 = disable all
y2 = grow -1
y3 = { y1;y2 }
```

Displaying Values with USE

If you enter:

```
use
```

at the Command window with no argument, the screen displays the names of all the directories in the current source file search path. For example, if your current search path only includes your current directory, a USE command without any options produces the following display:

```
Current search path: .
```

Saving Values with the WRITE Command

To save in a disk file the commands to establish all the current values of your terminal display parameters, aliases, source file search path, breakpoints, and tracepoints, enter:

```
write >> filename
```

at the Command window, where *filename* must be a valid filename. The Debugger creates an ASCII file called ***filename*.4db** or appends new commands to the existing file if there is already a file of that name in your current directory.

If you do not supply a filename, the information is saved under the *default filename*. This is the filename of the application program that you are currently analyzing, but with the extension **.4db**. The commands in this file are executed automatically whenever you use the Debugger with the same 4GL program. If you are not sure whether you want the current parameters reestablished automatically, you should supply a nondefault filename.

If your current debugging environment parameters all have default values, then an output file produced by a WRITE command with no options contains the following commands:

```
alias f1 = help
alias f2 = step
alias f3 = step into
alias f4 = continue
alias f5 = run
alias f6 = list break trace
alias f7 = list
alias  f8  =  dump
alias  f9  =  exit
use .
turn  on  autotoggle
turn off sourcetrace
turn on displaystops
turn  on  exitsource
turn off printdelay
timedelay  source      1
timedelay  command     0
source  lines          9
command  lines         10
list  display
```

If you use the ALIAS, APPLICATION DEVICE, GROW, TIMEDELAY, TURN, or USE commands to establish nondefault values, corresponding commands appear in the output file of the WRITE command.

Commands to establish any current breakpoints or tracepoints are also saved in the output file of a WRITE command when you specify no option. You can also use options of the WRITE command to prevent some of your debugging environment parameters from being saved. For example, WRITE ALIASES BREAK only saves commands to establish the current aliases and breakpoints. Read the syntax of WRITE at the end of Chapter 9 for more information.

Establishing Values with the READ Command

You can replace the current values of the terminal display parameters with other values by using a READ command to execute multiple Debugger commands from an ASCII file. To use this feature, enter:

```
read filename
```

at the command prompt, where *filename* is the name of an ASCII file that contains a list of Debugger commands. The input file must have the **.4db** extension, but this extension does not have to be included in READ commands.

The file that was created by a WRITE command in the previous section is an example of a file that could be used in a READ command to establish a set of terminal display parameters, aliases, breakpoints, and other debugging environment parameters. You can edit the file to include any other Debugger commands that are invoked by keywords, such as CALL or FUNCTIONS. The file cannot include commands that require control characters, such as **Interrupt**, **Screen**, or **Toggle**.

The commands in the file are executed in sequence when you invoke the READ command, just as if you were typing them at the command line. Their output appears in the Command window, but the commands themselves are not echoed there. Additional information about the READ command appears in the next section and in the summary of the syntax of READ in Chapter 9.

Establishing Parameters from Files

As noted in the last section, the READ command can be used to establish aliases, breakpoints, source file search paths, terminal display parameters, and tracepoints from a **.4db** file. This section presents additional information about these files, which can also be used to initialize your debugging environment without invoking READ.

Establishing System Default Parameters

Default aliases of the first nine function keys are specified in the *system initialization* file. This ASCII file contains the following ALIAS commands, which are executed automatically at the beginning of every debugging session:

```
alias f1 = run
alias f2 = step
alias f3 = step into
alias f4 = continue
alias f5 = help
alias f6 = list break trace
alias f7 = list
alias f8 = dump
alias f9 = exit
```

This file affects every user whose keyboard and **termcap** or **terminfo** file support function keys designated "f1, f2, ..., f9." The system initialization file is named **init.4db**, and it resides in the **$INFORMIXDIR/etc** directory. The **install** script created this file when you installed the Debugger.

Establishing User Default Parameters

The default values in the system initialization file are ignored by the Debugger if a *user initialization* file exists in your home directory. Commands in this file replace or modify any similar commands in the system initialization file.

Like the system initialization file, the name of the user initialization file must be **init.4db**. If you want to use this optional file, you must copy it to your home directory or create it there. Unlike the system initialization file, the user initialization file is not supplied by Informix, but you can create it with a text editor or by using the WRITE command of the Debugger.

Any Debugger commands that appear in this file are executed automatically at the beginning of every debugging session of every user who logs in under your account name.

Establishing Program Default Parameters

When you invoke the Debugger, the commands specified in the system or user **init.4db** files can be supplemented by additional commands in a *program initialization* file.

This file must have the same name as the compiled 4GL program that you are debugging, but with the extension **.4db**. If such a file exists in the same directory as the current 4GL program, its commands are executed automatically when you invoke the Debugger.

A program initialization file establishes the initial default debugging environment for a specific program. You can create this file with either a text editor or a WRITE command. The default filename in a WRITE command is the name of the program initialization file for the current program.

You can override the program initialization file default if you specify another initialization file when you invoke the Debugger from the command line.

Using Nondefault .4db Files

Besides the system, user, and program initialization files, whose commands can be automatically executed at the beginning of a debugging session, you can create other **.4db** files that include Debugger commands. As noted earlier in this chapter, entering a READ command and the filename at the Command window executes all of the commands in a **.4db** file.

You can also include the name of a **.4db** file in the command line that invokes the Debugger. You can do so by using the **-f** option of the **fgldb** command, as in the next example:

```
fgldb -f filename program
```

Here filename is a **.4db** initialization file, and *program* is the name of the **.4gi** or **.4go** application. Unless *filename* is in your current directory, you must prefix it with a pathname.

It is easy to create **.4db** files with the WRITE command. Debugging environment features that WRITE can save include those that follow:

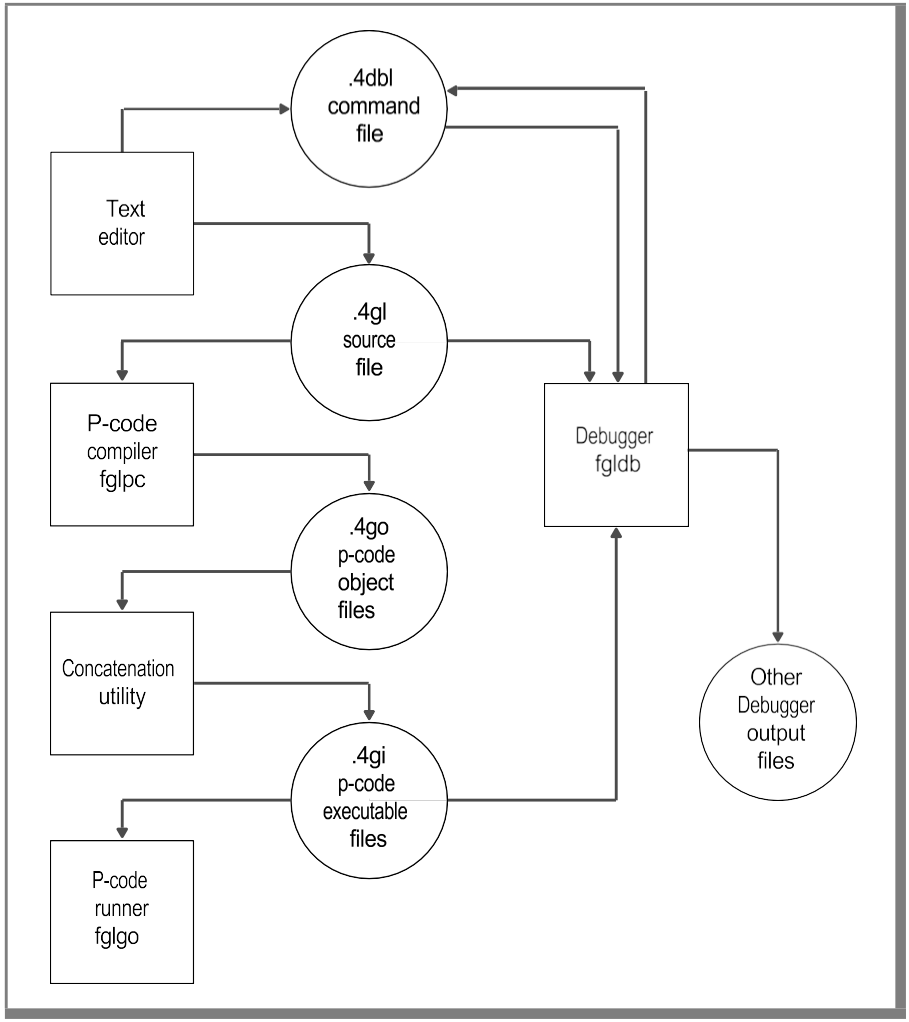
- Current aliases of your keyboard
- Current display parameters of your terminal
- Current source file search path
- Current breakpoints
- Current tracepoints

This allows you to take a *snapshot* of your current debugging environment and to reestablish the same parameters by using a single READ command in subsequent debugging sessions.

Because the **.4db** file is an ASCII file, you can also use a text editor to modify it according to the needs of a specific debugging task. These files save time at the start of a debugging session and enable you to develop a library of debugging environment templates for use in recurring tasks.

Figure 8-5 expands the schematic summary from Figure 8-3 to include files that can affect the debugging environment.

Figure 8-5
Input and Output Files in the Debugging Process



Exiting from the Debugging Environment

The EXIT command terminates the debugging session. You can only invoke it from the Command window.

When this is not your current window, press the **Interrupt** key (typically CTRL-C), if you are at the Application screen or at the Source window. If a Debugger HELP command is displaying messages, press RETURN until the Debugger screen appears. If you are using the Escape feature to execute an operating system command, terminate it, and press RETURN to display the Debugger screen. Now you can enter:

```
exit
```

This returns you to the prompt or menu from which you began the debugging session. If you invoked the Debugger in a command line, EXIT returns you to the system prompt. Unless you used a WRITE command before you ended the session, any changes that you made in the initial default values of your debugging environment no longer exist.

If you invoked the Debugger from the Programmer's Environment, you are prompted to press RETURN after an EXIT command. After you do this, the screen displays the menu from which you selected the **Debug** option. Here you can modify and recompile your 4GL source modules or perform other tasks.

If you exit to the Programmer's Environment inadvertently, simply press D at the current menu to choose the **Debug** option, and select the same 4GL program as your current application. This restores the same debugging environment that was current when you invoked the EXIT command.

If you select another program to debug, however, or if you return to the system prompt, the initial default values of the debugging environment parameters are in effect.

Chapter Summary

This chapter elaborated on the following points:

- You can invoke the Debugger from the Programmer's Environment or from the system prompt.
- You can specify parameters that control the following features of your debugging environment:
  - The 4GL program that the Debugger analyzes
  - The directory search path for 4GL source files
  - Keyboard aliases and function keys
  - The interactions of the Debugger screens and windows
  - Breakpoints to control program execution
  - Tracepoints to monitor program execution
- You can monitor a debugging session from the following screens and windows:
  - A Debugger screen that includes a Command window for Debugger commands and output, as well as a Source window to display source code of 4GL applications
  - An Application screen for output from the 4GL application

  Additional displays show help messages about Debugger commands and output from operating system commands.
- You can display the current debugging environment parameters by using the ALIAS, LIST, and USE commands.
- You can save the commands to establish the current debugging environment with a WRITE command.
- You can create files that automatically establish debugging environment parameters, or you can use a READ command to replace current parameters with values from a file.
- You can restore the same debugging environment from which you exited to the Programmer's Environment if you resume debugging the same program before you return to the system prompt.

Chapter

9

The Debugger Commands

In This Chapter

This chapter describes the command set of the Debugger. This chapter includes the following topics:

- A synopsis of Debugger control characters and commands, grouped by function
- Rules for specifying the scope of reference of INFORMIX-4GL variables
- Program execution status, and the distinction between active and inactive variables and functions
- The shortest unique forms of Debugger commands
- Typographic conventions for specifying command syntax
- Syntax of the Debugger commands, arranged in alphabetic order, with explanatory notes and examples of usage

The last section is intended as a reference guide to the individual commands, many of which have been illustrated in the earlier chapters.

Functionality of the Debugger Commands

The Debugger command set is small enough that if you work with the Debugger regularly, you will soon discover the logical relationships among its commands. If you are not yet familiar with the Debugger, however, it might be helpful to read this section, which groups the commands according to their function and briefly indicates what each command does. The Debugger commands and control characters are summarized under the following functional categories:

- Cursor movement keys and search commands
- Control keys for screen management
- Screen management commands
- Commands to display information
- Commands to control breakpoints and tracepoints
- Commands to specify values
- Commands for program execution

Cursor Movement Keys and Search Commands

Search commands and cursor movement keys can move the cursor within the Source and Command windows. They can also scroll the source code or the command buffer to lines that are not currently displayed in the window.

Search Commands and Wildcards

The slash (/) and question mark (?) keys allow you to search for a pattern within the Source or Command window (whichever is your current window). If the pattern is found, the cursor moves to the first line containing that pattern. You can use a question mark to search backward from the current cursor position or a slash to search forward.

A *pattern specification* is a string of no more than 50 blanks and characters, or up to 80 if a quotation mark (") is the first character. For example, the following command searches backward in the current window for the lowercase string `input`:

```
?input
```

A pattern specification can include wildcards and ranges of letters. The following symbols are for partial matches.

Symbol	Description
*	Matches any string of zero or more nonblank characters.
?	Matches any single nonblank character
[*d-p*]	Matches any character between *d* and *p* inclusive in the ASCII collating sequence, for $d < p$

For example, the search command:

```
/?[N-P]*T
```

searches forward, looking for a mixed-case or uppercase string such as `INPUT`, `cOnstrucT`, or `TOOLKIT` that matches the pattern.

If the Debugger finds the next instance of the pattern in a line that is not on the screen, the current window scrolls its display to include that line, and the cursor moves to it.

Unlike the cursor movement keys, which cannot scroll above the first line or below the last line of the command buffer or the source code module, a search command can *wrap around* the Source or Command window displays. This enables you to find a pattern in any line by searching in either direction.

Default Search Pattern

The first search command of a debugging session must include a pattern specification after the slash or question mark. In subsequent searches, the most recent pattern specification becomes the default. If you have searched the Source window since the last VIEW command, you can press RETURN to repeat the most recent search command from the current cursor position. The Debugger then searches in the same direction for the same pattern.

Cursor Movement at the Debugger Screen

If the Command or Source window is your current window, the following keys move your cursor to a different line.

Key	Effect
CTRL-K	(Or UP ARROW) moves up one line
CTRL-J	(Or DOWN ARROW or RETURN) moves down one line
CTRL-B	Moves up one full window, less one line
CTRL-F	Moves down one full window, less one line
CTRL-U	Moves up one-half window
CTRL-D	Moves down one-half window

If you prefix with a number any of the arrow keys or control characters in this list, the Debugger repeats the command as many times as you specify. For example, typing `3` and then pressing CTRL-U moves the cursor up one-and-a-half windows or to the first line, whichever comes sooner.

When you use the Escape feature or a search command at the Source window, or specify a command line at the Command window, pressing BACKSPACE (or CTRL-H or LEFT ARROW) moves the cursor one space to the left. This also deletes any character that you typed there but had not yet entered.

Features of the Command Window Cursor

At the Command window, you can type a nonnegative number and press RETURN if your cursor is above the last $ prompt. This moves your cursor down by the specified number of lines or to the current $ prompt, whichever comes first. For example, entering `20` moves the cursor to the 20th line below its current position (or to the current $ prompt if that is 20 or fewer lines below the cursor position).

If the cursor is at an earlier line of the command buffer, typing any alphabetic character moves the cursor to the $ prompt and echoes that character.

Features of the Source Window Cursor

The Source window recognizes three cursor movement commands that have no effect (or a different effect) at the Command window.

Command	Effect
nline	Moves to line number *nline*, where *nline* is an integer
$	Moves to the last line of the current module
RETURN	Moves to the next instance of the most recent search pattern since the last VIEW command

For example, entering `20` at the Source window moves the cursor to line number 20 of the current module, displaying part of the 4GL source file that contains that line.

Cursor Movement in Help and Application Displays

Search commands cannot search for patterns within help messages or in 4GL program output on the Application screen. You can use cursor movement keys, however, to move your cursor or to highlight options of Help or Application screen menus.

When your screen prompts you to select a topic from the help facility, you can use the arrow keys or BACKSPACE to move the cursor and highlight help options. You can also use these keys or the SPACEBAR to toggle between the **Screen** and **Resume** options above specific help messages. (Pressing RETURN selects the currently highlighted option.)

The same keys can also be used at the Application screen when your 4GL program requires keyboard input. This is only possible when the last 4GL statement is a PROMPT, INPUT, INPUT ARRAY, CONSTRUCT, or MENU statement. You cannot move the cursor within the Application screen if 4GL output is being displayed by a Toggle command.

Control Keys for Screen Management

You can use the following control characters to manipulate the displays of the Debugger windows and screens or to select the current window.

Key	Effect
CTRL-C	(Or DEL or whatever is your **Interrupt** key) switches the current window from the Application screen or Source window to the Command window. If a 4GL program is running, its execution is suspended.
CTRL-P	Copies the current display of the Application screen or Debugger screen to a disk file. This is called the **Screen** key.
CTRL-Q	Enables the keyboard and screen for terminal I/O after you press CTRL-S. This is called the **X-ON** key.
CTRL-R	Redraws the screen display. This is called the **Redraw** key.
CTRL-S	Disables terminal I/O, usually to prevent information from scrolling off the screen before you can read it. This is called the **X-OFF** key.
CTRL-T	Switches from the Debugger screen to the Application screen. To restore the Debugger screen, press this **Toggle** key again.

After the **Toggle** key switches your display to the Application screen, the Debugger interprets any subsequent keystroke (except these control keys) as beginning a command at your previous current window.

Screen Management Commands

Except for the ! command, by which you can enter an operating system command from the Source or Command window, the remaining Debugger commands can only be invoked from the Command window. The following commands affect the terminal display.

Command	Purpose
! *command*	Executes an operating system command from the Source or Command window, displaying any output. This is the *Escape* feature.
APPLICATION DEVICE	Redirects screen output of the current 4GL program or function to another terminal.
GROW	Specifies the relative sizes of the Command and Source windows.
HELP	Temporarily replaces the Debugger screen with Debugger Help facility output.
TIMEDELAY	Specifies the speed at which successive 4GL statements in the Source window are highlighted when the SOURCETRACE parameter is ON. It can independently specify the speed at which the Command window displays lines of Debugger output.
TURN	Specifies whether terminal display parameters AUTOTOGGLE, DISPLAYSTOPS, EXITSOURCE, PRINTDELAY, and SOURCETRACE are ON or OFF.
VIEW	Specifies the 4GL function or module that the Source window displays and makes the Source window the current window.

The terminal display parameters are described in Chapter 8, "The Debugging Environment," and in the syntax descriptions of their respective commands.

Commands to Display Information

Other commands enable you to display information about the current debugging session in the Command window or to redirect output from Debugger commands to a disk file.

Command	Purpose
DUMP	Displays values of program variables on the screen or saves them in a file
FUNCTIONS	Displays the names of programmer-defined 4GL functions in the current program
LIST	Displays the current terminal display parameters, breakpoints, and tracepoints
PRINT	Displays on the screen the value of an expression or copies it to a file
VARIABLE	Displays the declaration of a variable, record, or array on the screen or copies it to a file
WHERE	Displays on the screen or copies to a file the names of the functions that have been called (but have not returned) before the current 4GL statement executed
WRITE	Saves in a disk file the Debugger commands necessary to establish the current values of the aliases, display parameters, breakpoints, tracepoints, and source file search path

The previous sections described the HELP command (to display command syntax) and the **Screen** key (to save in a file the current Application screen or Debugger screen).

The ALIAS and USE commands are described in "Commands to Specify Values" on page 9-14 . These commands can respectively display your current keyboard aliases and your source file search path.

Commands to Control Breakpoints and Tracepoints

Some Debugger commands enable you to examine the logic of a 4GL program by specifying breakpoints and tracepoints and the conditions under which these points are active. If a statement is marked by a breakpoint, execution stops when the statement is reached a specified number of times. If a statement is marked by a tracepoint, information is displayed when the statement is executed. You can also deactivate or delete breakpoints or tracepoints.

Command	Purpose
BREAK	Establishes a breakpoint and optionally specifies the conditions under which the breakpoint suspends program execution
DISABLE	Deactivates a breakpoint or tracepoint without removing it or deactivates all such points in a function or program
ENABLE	Reactivates a breakpoint or tracepoint that had been disabled or reactivates all such points in a function or program
NOBREAK	Deletes an existing breakpoint or all the breakpoints in a function or program
NOTRACE	Deletes an existing tracepoint, or all the tracepoints in a function or program
TRACE	Specifies a tracepoint to indicate when a 4GL statement or function executes or when a variable changes during program execution

These breakpoint and tracepoint commands are among the most important features of the Debugger when you are attempting to identify logical flaws in a program. They are also helpful when you are analyzing a 4GL program that was written by someone else.

Commands to Specify Values

Some of the Debugger commands can enhance the efficiency of a debugging session by assigning values to function keys or 4GL program variables, or performing certain housekeeping tasks without exiting from the Debugger.

Command	Purpose
ALIAS	Assigns names or function keys to command strings so that frequently used commands can be entered in a few keystrokes
CLEANUP	Reinitializes all variables, forms, and windows so that you can repeatedly invoke CALL commands to restart the same 4GL function
DATABASE	Specifies the current database and closes any previously open database
LET	Assigns a value to a variable so that you can change the current values of variables in a 4GL program that you are debugging
READ	Executes multiple Debugger commands from a file so that you can reestablish the environment of an earlier debugging session that you saved by a WRITE command
USE	Specifies a new source file search path so that you can access 4GL files that are not found in any directory of your current search path

You do not need to issue a CLEANUP command before RUN. Do not invoke LET immediately before a RUN command (described in the next group of commands) because RUN automatically reinitializes all program variables with zero or null values and closes all forms and windows in the 4GL program.

Commands for Program Execution

The last group of Debugger commands enables you to control the execution of 4GL statements within the debugging environment or to exit from the Debugger.

Command	Purpose
CALL	Executes a 4GL function, or executes a C or INFORMIX-ESQL/C function linked to the program
CONTINUE	Resumes execution of a suspended program or sends an Interrupt signal or a Quit signal to a currently executing 4GL program
EXIT	Terminates execution of the Debugger, returning you to the Programmer's Environment or to the operating system
RUN	Initializes and executes the 4GL program within the debugging environment
STEP	Executes one or more individual 4GL statements

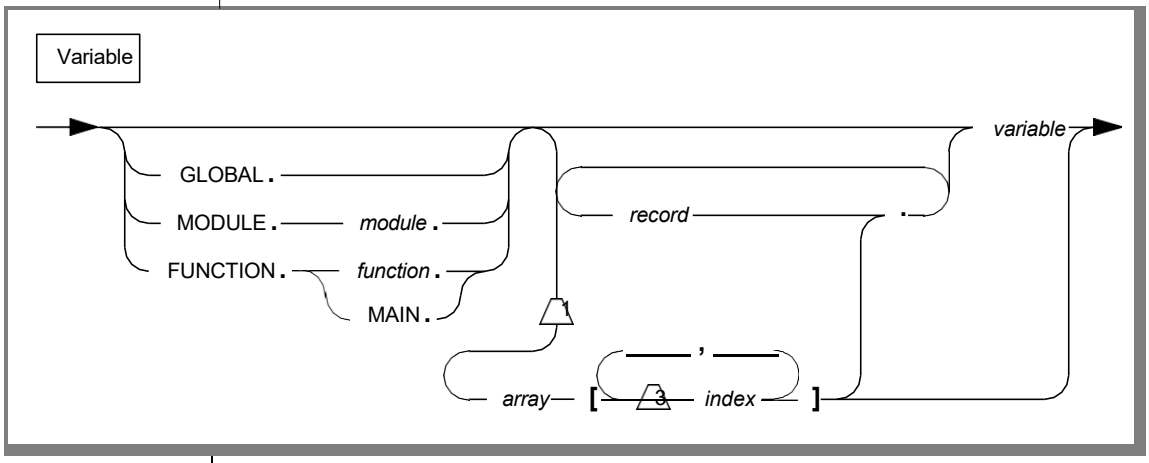
The earlier sections on screen management described the **Interrupt** key (often CTRL-C or DEL) and the Escape feature that the exclamation point (!) invokes. The first suspends execution of a running 4GL application. The Escape feature can execute an operating system command line.

This concludes the descriptions of functionally similar groups of commands. Several brief sections that follow describe topics that affect Debugger commands, including the scope of reference of variables, active versus inactive variables and functions, and abbreviations for Debugger keywords. The final section of this chapter provides detailed information and examples of each Debugger command.

Scope of Reference

Some Debugger commands can use *qualifiers* to identify variables outside the current function or to distinguish between similarly named variables in a 4GL program, module, or function. A global variable retains its value throughout a program. Other variables can be restricted in their scope to a single module or function.

You can qualify the name of a program variable with prefixes to indicate its scope of reference. A period (.) separates a qualifier from the name of the variable or from another qualifier. Use the following syntax to qualify a variable.



Element	Description
array	The identifier of a 4GL array of records in which *variable* is a member or of an array whose element you specify in a **Debugger** command.
function	The identifier of the function in which variable is declared.
index	A literal integer specifying an index within a dimension of array.
module	The filename of the source-code module in which variable is declared.
record	The identifier of the 4GL record of which *variable* is a member or of a record whose members include a record that has *variable* as a member.
variable	The unqualified identifier of a 4GL program variable.

When using qualifiers, take the following considerations into account:

- If GLOBAL is the first qualifier, the variable is among your globals, and the next identifier must be the variable name or the name of a record or array that includes the variable. For example, **global.x** specifies a *global* variable **x**.
- If MODULE is the first qualifier, the next must be the name of the module where the variable is defined. For example, **module.code.x** specifies a *module* variable that is declared in a module called **code**.
- If FUNCTION is the first qualifier, the next must be MAIN or the name of the function where the variable is defined. For example, **function.main.x** specifies a *local* variable that is declared in the **main** program block.

Here the meaning of the unqualified name **x** depends on the context (that is, on your current 4GL function).

The Scope of Reference Rules

This section outlines how variable names in Debugger commands are identified with specific 4GL program variables. You specify a variable name at the Command window. (Here *variable* symbolizes the identifier of any 4GL variable.) The Debugger then searches for a corresponding 4GL variable by sequentially applying the following rules:

1. Is the specification GLOBAL.*variable*?
2. Is the specification MODULE.*module-name.variable*?
3. Is the specification FUNCTION.*function-name.variable*?
4. Is there a matching name among the *local* variables that are defined in the current function or program block?
5. Is there a matching name among the *module* variables that are defined in the current module?
6. Is there a matching name among the *global* variables that are declared in a GLOBALS statement?

Throughout this manual, the terms *current function* and *current module* refer to the function or module in the Source window or (if the DISPLAYSTOPS parameter is OFF) the currently executing function or module. The VIEW command can specify the current function.

As these rules imply, the Debugger does not search for a variable outside the current module unless the variable is a global or module variable or unless you specify the GLOBAL, MODULE, or FUNCTION qualifier.

These rules for qualifying variables are features of the Debugger that are not supported by 4GL. You can prefix variable names with the GLOBAL, MODULE, or FUNCTION qualifier in Debugger commands but not in 4GL statements. Error messages appear if you attempt to compile **.4gl** files that contain any of these qualifiers. For example, suppose that in a PRINT command you specified a qualified variable:

```
print  cust.ord.name
```

Before the Debugger can execute the command, it must first identify the variable *cust.ord.name*. Although its multiple qualifiers indicate that the variable **name** is a member of a record within a record, the Debugger applies the same rules that it would apply to the name of a variable that had no qualifiers.

Because the first qualifier is neither GLOBAL, MODULE, nor FUNCTION, the search for a variable called **cust.ord.name** would proceed in the following sequence:

1. Look for a record **cust** containing a member record **ord** with a member **name** in the current function.
2. Look for a record **cust** containing a member record **ord** with a member **name** in the current module.
3. Look for a record **cust** containing a member record **ord** with a member **name** among the global variables.

The search stops when a variable is found. If no variable is found that matches your specification, an error message appears, and the $ prompt returns.

Example of Qualifying Variables

The next example illustrates how qualifiers can distinguish among program variables that have the same name but different scopes of reference.

4GL does not allow a module variable and a global variable that are both defined in the same module to have the same name. In the example that follows, the global variable marked { 1 } and the module variable marked { 2 } cannot have the same name.

Suppose that you are debugging the following program, whose variables (defined in lines marked by numbers in braces) all have similar names:

```
 1   GLOBALS
 2        DEFINE j INTEGER                    {1}
 3   END GLOBALS
 4
 5   DEFINE i INTEGER                         {2}
 6
 7   MAIN
 8        DEFINE i INTEGER                    {3}
 9       CALL b()
10   END MAIN
11
12   FUNCTION a()
13        DEFINE i INTEGER                    {4}
14        DEFINE j INTEGER                    {5}
15       DEFINE b RECORD
16           i INTEGER                        {6}
17       END RECORD
18   END FUNCTION
19
20   FUNCTION b()
21        DEFINE i INTEGER                    {7}
22       CALL a()
23   END FUNCTION
```

Assume that this module is called **mod.4gl** and that a breakpoint has stopped execution in function **a**. Then the Debugger commands listed at the left reference the following variables.

Command	Variable	Description
`print i`	{4}	Variable **i** in current function **a()**
`print b.i`	{6}	Member **i** of record **b** in function **a()**
`print FUNCTION.b.i`	{7}	Variable **i** in function **b()**

(1 of 2)

Command	Variable	Description
`print FUNCTION.main.i`	{3}	Variable **i** in main program block
`print MODULE.mod.i`	{2}	Variable **i** in current module
`print GLOBAL.j`	{1}	Global variable j
`print j`	{5}	Variable **j** in current function **a()**

(2 of 2)

Here the scope of reference of **i** and **j** depends on your current 4GL statement, function, and module. If another function were the current function in this example, the first two commands would not refer to the variables marked {4} and {6} but to different variables. The next section, "Active Functions and Variables," describes another way in which some Debugger commands are context sensitive.

Active Functions and Variables

This section identifies the commands that require *active functions* or *active variables* in the current 4GL application. Whether any functions or variables are active depends upon the current state of program execution.

The Status of Program Execution

When the Debugger can accept your commands at the Command window, execution of the 4GL program is either suspended or terminated abnormally, or else the program is not running. The following circumstances determine which of these states is your current state of execution:

- **Suspended**. Program execution can be suspended by any of these:
  - An active breakpoint
  - The Interrupt key
  - A STEP command that has executed its 4GL statements

- **Terminated abnormally.** Execution can be terminated abnormally by any of these:
  - A fatal error
  - A CONTINUE INTERRUPT command
  - A CONTINUE QUIT command
  - A 4GL EXIT PROGRAM statement
- **Not running.** The 4GL program is not running if any of the following conditions describes your current session:
  - No RUN or CALL command has begun execution.
  - CLEANUP followed the most recent RUN or CALL command.
  - Execution terminated normally.

If a 4GL EXIT PROGRAM statement is encountered, the Debugger treats program execution as abnormally terminated. But CONTINUE INTERRUPT or CONTINUE QUIT commands do not result in abnormal termination unless your 4GL source code fails to handle these signals.

Program execution begins with a valid RUN or CALL command and resumes after a STEP or CONTINUE command. Unless execution is currently suspended, the Command window displays an error message after any CONTINUE or STEP command.

Active Functions

An *active function* is a 4GL function that has been called but has not yet returned. There are no active functions unless execution of the 4GL program has begun but is suspended or else has terminated abnormally. When the 4GL program is not running, there is no active function.

The WHERE command displays the names of all currently active functions. Error messages appear after WHERE, DUMP, and LET commands when no function is active.

The Debugger always treats the MAIN section of a 4GL program as a function.

Active Variables

An *active variable* is any global variable or any 4GL program variable in an active function. There are no active variables unless CALL or RUN has begun execution. The Debugger displays an error message if a CALL, LET, or PRINT command references a variable that is not active.

Even if the 4GL program is not running, you can reference the name of any global variable as an argument of a function in a CALL command. You must substitute appropriate constants, however, for the names of any inactive variables in CALL, PRINT, and LET commands.

Examples of Inactive Functions and Variables

Suppose that your current program is the same **mod.4gl** program that appeared earlier in this chapter to illustrate the scope of reference of variables:

```
    1   GLOBALS
    2         DEFINE j INTEGER                    {1}
    3   END GLOBALS
    4
    5   DEFINE i INTEGER                          {2}
    6
    7   MAIN
    8         DEFINE i INTEGER                    {3}
>   9        CALL b()
   10   END MAIN
   11
   12   FUNCTION a()
   13         DEFINE i INTEGER                    {4}
   14         DEFINE j INTEGER                    {5}
   15        DEFINE b RECORD
   16            i INTEGER                        {6}
   17        END RECORD
>  18   END FUNCTION
   19
   20   FUNCTION b()
   21         DEFINE i INTEGER                    {7}
>  22        CALL a()
   23   END FUNCTION
```

Suppose that you have not yet issued a RUN or CALL command in this debugging session. If you enter a WHERE command, the Command window displays this message:

```
-16387: Program is not currently being executed.
```

If you enter a CONTINUE, DUMP, or STEP command, similar error messages appear because program execution has not begun.

If no breakpoints exist in this program, these commands produce the same effect after a RUN or CALL command because execution terminates normally, leaving no function active. After normal termination, for example, a DUMP command produces the following error message:

```
-16362: No active function.
```

When the 4GL program is not running, CONTINUE and STEP commands only produce error messages, and LET and PRINT commands cannot include the names of any program variables. No local variable or module variable can be referenced in a CALL command if no function is active.

For purposes of illustration, suppose that you set active breakpoints at lines 9, 18, and 22 with BREAK commands. If you begin execution with a RUN command, the program stops at the breakpoint in line 9. Now a WHERE command displays the only active function:

```
main() at line 9 in mod.4gl
```

Entering DUMP ALL displays the names and values of all the active variables, which are in the lines marked {1}, {2}, and {3}.

You can resume execution by entering `CONTINUE`, `STEP`, or `CALL b()`. The breakpoint at line 22 suspends execution before function **a** is called. This time a WHERE command lists two active functions:

```
b() at line 22 in mod.4gl
main() at line 9 in mod.4gl
```

Entering DUMP ALL shows that the variables defined in the lines marked {1}, {2}, {3}, and {7} are active.

Entering the `CONTINUE`, `STEP`, or `CALL a()` command resumes execution, but the breakpoint in line 18 stops the program before any function returns. Now WHERE and DUMP commands would show that all the functions and variables are active.

As this example shows, your choice of breakpoints might be influenced by the fact that some command arguments are restricted to active functions or to active variables.

In summary, error messages can appear after CALL, CONTINUE, DUMP, LET, PRINT, STEP, or WHERE commands when no function is active, or when a command references an inactive variable.

Other Debugger commands, including BREAK, TRACE, FUNCTIONS, and VARIABLE, make no distinction between active and inactive functions or variables. If your breakpoints or tracepoints specify additional commands to execute, however, any variables that they reference must be active when the breakpoint or the tracepoint is reached. For example, if you enter the command:

```
break (fend) x { print y; print function.fz.z }
```

then **y** must be an active variable, and **fz** must be an active function when the value of **x** changes in **fend**.

Understanding the distinction between active and inactive functions and variables can save you time by avoiding errors. The Debugger error messages are described in the "Error Messages" section near the end of this manual.

Short Forms of Keywords

The Debugger can recognize a command keyword (and most options) if you truncate the complete keyword from the right. The abbreviated form must be unambiguous.

The following table contains the four Debugger commands that derive their shortest unique forms from default options that are implied when you do not enter an alternative option.

Command and Default	Shortest Unique Form
APPLICATION DEVICE	ap
GROW SOURCE	**g**
TIMEDELAY SOURCE	**ti**
TURN ON	**tu**

Command options *cannot* be abbreviated, however, if the name of a program, function, or variable could appear in the same position as the option. For example, the **f** in the command:

```
tr f
```

cannot be a valid abbreviation for the FUNCTIONS option of the TRACE command because it could also be interpreted as the name of a variable or function.

The complete form (uppercase) and the shortest unique form (lowercase) of each command keyword and its options appear alphabetically in the list on the pages that follow. Most commands cannot be executed unless you include additional arguments.

Six commands are invoked by nonalphabetic characters rather than by keywords. You can use the names ESCAPE, INTERRUPT, REDRAW, SCREEN, SEARCH, and TOGGLE, if prefixed by an underscore(_), as HELP command options. For example, to display information about the **search** command, you can enter an abbreviated or complete form of:

```
help _search
```

The following table lists the Debugger command keywords.

Command	Option	Shortest Form
ALIAS		**al**
APPLICATION		**ap**
	DEVICE	**ap**
BREAK		**b**
	IF	**b if**
CALL		**ca**
CLEANUP		**cl**
	ALL	**cl a**
CONTINUE		**co**
	INTERRUPT	**co i**
	QUIT	**co q**
DATABASE		**da**

(1 of 5)

Command	Option	Shortest Form
DISABLE		**di**
	ALL	**di all**
DUMP		**du**
	ALL	**du a**
	GLOBALS	**du g**
ENABLE		**en**
	ALL	**en all**
EXIT		**ex**
FUNCTIONS		**f**
GROW		**g**
	COMMAND	**c**
	SOURCE	**g**

(2 of 5)

Command	Option	Shortest Form
HELP		**h**
	ALIAS	**h al**
	ALL	**h all**
	APPLICATION DEVICE	**h ap**
	BREAK	**h b**
	CALL	**h ca**
	CLEANUP	**h cl**
	CONTINUE	**h co**
	DATABASE	**h da**
	DISABLE	**h di**
	DUMP	**h du**
	ENABLE	**h en**
	_ESCAPE	**h _e**
	EXIT	**h ex**
	FUNCTIONS	**h f**
	GROW	**h g**
	HELP	**h h**
	_INTERRUPT	**h _i**
	LET	**h le**
	LIST	**h li**
	NOBREAK	**h nob**
	NOTRACE	**h not**
	PRINT	**h p**
	READ	**h re**
	_REDRAW	**h _r**
	RUN	**h ru**
	_SCREEN	**h _sc**
	_SEARCH	**h _se**
	STEP	**h s**
	TIMEDELAY	**h ti**
	_TOGGLE	**h _t**
	TRACE	**h tr**
	TURN	**h tu**
	USE	**h u**
	VARIABLE	**h va**
	VIEW	**h vi**
	WHERE	**h wr**
	WRITE	
LET		**le**

Command	Option	Shortest Form
LIST		**li**
	BREAK	**li b**
	DISPLAY	**li d**
	TRACE	**li t**
NOBREAK		**nob**
	ALL	**nob all**
NOTRACE		**not**
	ALL	**not all**
PRINT		**p**
READ		**re**
RUN		**ru**
STEP		**s**
	INTO	**s i**
	NOBREAK	**s n**
TIMEDELAY		**ti**
	SOURCE	**ti**
	COMMAND	**ti c**
TRACE		**tr**
	FUNCTIONS	**tr functions**
TURN ON		**tu**
	AUTOTOGGLE	**tu a**
	DISPLAYSTOPS	**tu d**
	EXITSOURCE	**tu e**
	PRINTDELAY	**tu p**
	SOURCETRACE	**tu s**
TURN OFF		**tu of**
	AUTOTOGGLE	**tu of a**
	DISPLAYSTOPS	**tu of d**
	EXITSOURCE	**tu of e**
	PRINTDELAY	**tu of p**
	SOURCETRACE	**tu of s**
USE		**u**

(4 of 5)

Command	Option	Shortest Form
VARIABLE		**va**
	ALL	**va all**
	GLOBALS	**va globals**
VIEW		**vi**
WHERE		**wh**
WRITE		**wr**
	ALIAS	**wr a**
	BREAK	**wr b**
	DISPLAY	**wr d**
	TRACE	**wr t**

(5 of 5)

Conventions for Command Syntax Notation

The typographic conventions identified in this section are used to define the syntax of the Debugger commands. The examples that illustrate these conventions are from actual Debugger commands. For simplicity and clarity, however, many of these examples present only part of the actual command syntax. See "Syntax of the Debugger Commands" on page 9-32 for the complete syntax of all the commands.

Capital Letters

Strings in uppercase letters are required keywords or options that you enter as shown. You can substitute equivalent shortened forms or lowercase. For example:

```
HELP  TIMEDELAY
```

means to enter the command `help timedelay`.

Italics

Strings in *lowercase italic* letters are terms for which you must substitute a specific identifier or value. For example:

```
DATABASE database-name
```

means to enter the name of some specific database after the keyword `database`. The explanation and notes that follow the syntax describe the class of terms.

Brackets

Brackets enclose terms that are not required. Do not include such brackets in a command line (unless you are specifying an element of an array). For example:

```
APPLICATION [DEVICE]
```

means that you must enter `application`, but you also have the option of entering `application device`. Each set of brackets represents a different condition. If several terms in separate brackets follow a required keyword, you can use all, any, or none of them. For example:

```
LIST [BREAK] [TRACE] [DISPLAY]
```

means that you can enter `list` alone, `list break`, or `list` followed by any combination of the three options.

Brackets nested within brackets indicate dependencies among options. You can only use the option in the inner brackets if you select the option in the outer brackets. For example:

```
BREAK [[module.]lineno]
```

means that you cannot specify a *module.* prefix unless you also specify a value for *lineno*.

Pipe Symbol

If a vertical bar, or pipe symbol, separates terms within the same set of brackets, you can choose no more than one of those options. Do not include any vertical bars in a command line. For example:

```
CONTINUE [INTERRUPT | QUIT]
```

means that you can enter `continue` alone, or else `continue` followed by `interrupt` or by `quit` but not by both.

Braces

Braces around a list of options means that you must choose one of the options. Do not include such braces (or the brackets or the vertical bars that separate the listed options) in a command line. For example:

```
TURN [ON | OFF] {AUTOTOGGLE | DISPLAYSTOPS |
     EXITSOURCE | PRINTDELAY | SOURCETRACE} ...
```

means that you must enter at least one of the options from the list in braces that begins with AUTOTOGGLE and ends with SOURCETRACE. (The three dots at the end mean that you can include additional options from this list.)

There is an exceptional use of braces. You must include braces around any command strings that you include in an ALIAS, BREAK, or TRACE command. This is indicated by the notation { *commands* } or { *cmd_str*... } in their syntax.

Underscore

In lists of options within brackets, the underscore marks the default option. In the previous example, the underscore means that ON is the default, so if you invoke the TURN command without specifying OFF or ON, the Debugger selects ON as your option. (The vertical separator indicates that you cannot select both ON and OFF in the same TURN command.)

Ellipsis Points

Ellipsis points within brackets mean that you have the option to repeat a term like the previous term. Do not include such dots in a command line. For example:

```
CALL function ([arg [, arg...]])
```

means that if you include an optional argument *arg* of a function, you can also include additional arguments, separated by commas.

In examples of usage, ellipsis points alone on a line mean that an indefinite number of other commands can occur between commands above and below the dotted line.

All other special symbols, such as " (!) . * - > = ? / ' , should be interpreted literally, rather than as conventional symbols. You should enter these symbols in your command line exactly as they appear in the syntax descriptions of the next section.

Except for the *literal braces* around command strings in the syntax of ALIAS, BREAK, and TRACE, these are the same typographic conventions that are used in the 4GL documentation to represent the syntax of 4GL statements.

Syntax of the Debugger Commands

The rest of this chapter describes the syntax of the commands that the Debugger recognizes at the $ prompt of the Command window, with explanatory notes and selected examples of usage.

Commands are listed in alphabetical order, according to the keyword that invokes each command. Six exceptions (nonkeyword commands) are as follows.

Name	Invoked By	Syntax Heading
Escape feature	!	ESCAPE
Interrupt	CTRL-C	INTERRUPT
Redraw	CTRL-R	REDRAW

(1 of 2)

Name	Invoked By	Syntax Heading
Screen	CTRL-P	SCREEN
Toggle	CTRL-T	TOGGLE
Search	/ or ?	SEARCH

(2 of 2)

A **.4db** file cannot include any of these nonkeyword commands. These are also the only commands that you can invoke at both the Source window and the Command window. The Interrupt, Redraw, Screen, and Toggle commands are the only commands that you can execute at the Application screen. You can enable or disable terminal I/O at any screen or window with the X-ON (CTRL-Q) or X-OFF (CTRL-S) keys.

Specific Restrictions on Debugger Commands

The Debugger imposes the following limits on the maximum length or complexity of command lines:

- Up to 256 characters in a command line
- Up to 80 characters in a quoted string
- Up to 70 characters in a pathname
- Up to 50 characters in a nonquoted search pattern
- Up to 10 nested READ commands
- Up to 5 nested ALIAS commands

Multiple Command and Continuation Symbols

Two special symbols can be used in Debugger command lines.

Key	Effect
\	Backslash is the *continuation symbol* for commands that require more characters than fit in one line. This cannot divide keywords or strings but can appear where a blank would occur.
;	Semicolon (like RETURN) isa *command separator*. You can use these to enter several commands in the same line.

For example, you could type at the Command window:

```
print x;   print y;   print z
```

When you press RETURN, 4GL program variables **x**, **y**, and **z** are evaluated in succession.

You can use both a semicolon and a backslash in the same line. For example, you could type at the Command window:

```
list d; call item_total(); turn off autotoggle \
displaystops exitsource sourcetrace; view
```

When you press RETURN, the LIST, CALL, TURN, and VIEW commands are executed in succession.

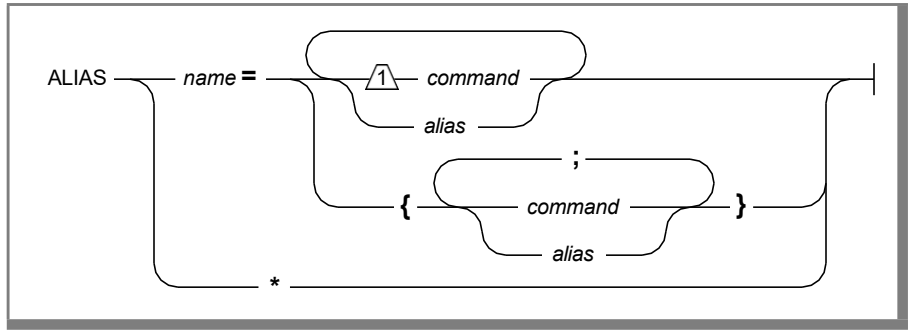
Semicolons and backslashes can be in the specifications of ALIAS, BREAK, and TRACE commands. For example:

```
break * (dfunc) "bi" -6 if latex[i] { print \
"This is my big break."; print latex[i]; where }
```

The backslash at the end of the first line enables you to enter a BREAK command that is too long to fit in a single line.

ALIAS

Use ALIAS to assign a name or function key to a command string.



Element	Description
alias	The name of an alias that was declared in a previous ALIAS command. (The number of *alias* terms is not restricted, but *name* cannot be defined by other aliases that are nested more than five layers deep.)
command	A string that contains all (or the first part) of a Debugger command. Search, Escape, and control-key commands are not valid here.
name	The name of a function key (f1, f2, ...) or an alphanumeric string that is defined here as an alias for the commands that follow the = symbol. The *name* cannot be the complete name of another Debugger command.

On some terminals you can use a function key name, such as `f1` for F1, as an alias. Function key names must be defined in the file that is specified by the **TERMCAP** environment variable or in the **terminfo** directory specified by the **TERMINFO** environment variable.

The command string that you assign to an alias can be a command or the first part of a command, or it can group several commands enclosed by a single pair of braces ({ }). Use a semicolon (;) to separate commands, or put each command on a new line.

Entering the alias or pressing the function key sends the command string to the Debugger. The Command window echoes the alias and then displays the expanded command, substituting command strings for the alias before executing the command. If any of the expanded command strings themselves contain aliases, the Debugger repeats the process of expanding aliases before executing the expanded command.

The asterisk (*) option displays all your current aliases.

Warning: *If you use the terminal* ***Setup*** *keys to program your function keys, the Debugger might not recognize your function keys.*

An alias persists until another ALIAS command reassigns the same name or function key or until an EXIT command ends the debugging session. If you exit to the Programmer's Environment and then resume debugging the same 4GL program without returning to the operating system, your aliases are restored. Otherwise, they are replaced by the initial default aliases.

Examples

If you have not established any aliases, you can enter:

```
alias *
```

to display the default aliases that are defined by the system file **init.4db** for the first nine function keys:

```
alias f1 = help
alias f2 = step
alias f3 = step into
alias f4 = continue
alias f5 = run
alias f6 = list break trace
alias f7 = list
alias f8 = dump
alias f9 = exit
```

To establish **x** as an alias for EXIT, enter:

```
alias x = exit
```

The next example assigns two commands to function key F9:

```
alias f9 = { print add_flag; view }
```

This is equivalent to the following command:

```
alias f9 = { print add_flag view }
```

You can invoke both commands by pressing the F9 key or by entering `f9`.

The command:

```
alias y1 = { f8; f3 main; f6 }
```

enters whatever commands the F3, F6, and F8 function keys currently alias, with **main** the argument of the second command. When you enter `y1`, the Command window echoes the command string that you specified within the braces and then displays the expanded string by substituting it for the aliases. If the expanded commands are valid, they are then executed in the order in which they were specified in the ALIAS command.

Related Commands

WRITE

APPLICATION DEVICE

Use APPLICATION DEVICE to redirect screen output from the 4GL application to a second video terminal.

Element	Description
device	The name of a terminal device to display the Application screen. The device specification must include the complete terminal pathname of the device.

This command enables you to monitor the Application screen continuously, rather than only when your screen is not displaying the Debugger screen or the Help screen.

This command can be useful if you are working on a multiuser system and have access to two terminals that can support identical **termcap** or **terminfo** entries. The terminal specified in this command uses the **termcap** or **terminfo** entry of the terminal that invoked the Debugger.

When the application program requires input, only the keyboard of the terminal that invoked the Debugger can enter input. Because the second terminal is only used for output, its keyboard is not enabled until you invoke the EXIT command from the terminal that is running the Debugger.

Both terminals must be logged in under the same account name.

Do not choose the terminal that is running the Debugger as your application device.

Important: *If you invoke the APPLICATION DEVICE command, you must specify another terminal. Otherwise you will have difficulty reading screen output, and your keyboard might lock up.*

The **Toggle** key and the AUTOTOGGLE option of the TURN command have no effect after an APPLICATION DEVICE command. The **Interrupt** key switches keyboard input to the Command window if the 4GL program is requesting input.

You can display the terminal pathname of any logged-in terminal by using its keyboard to enter the command `tty`.

Example

The following command selects a terminal designated as **/dev/tty05** to be the device to display screen output from the 4GL application program being debugged:

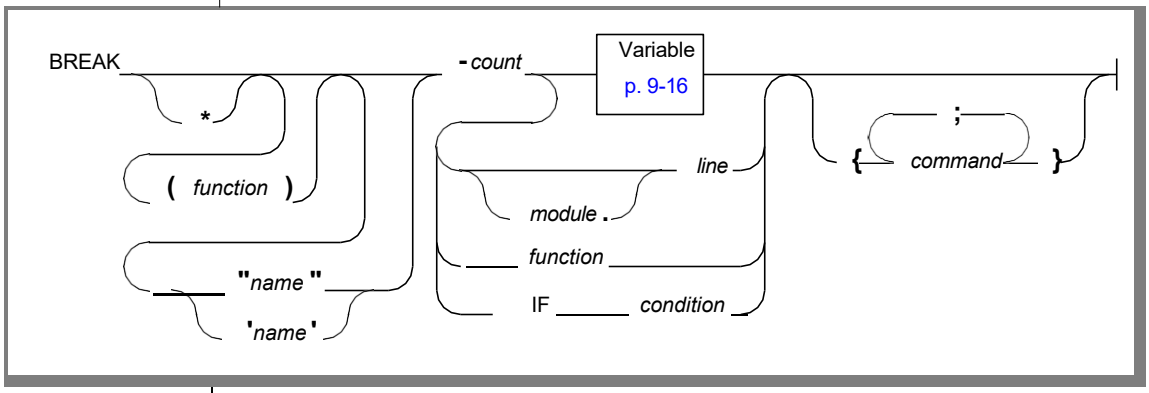
```
app dev /dev/tty05
```

Related Commands

EXIT, INTERRUPT, LIST, WRITE

BREAK

Use BREAK to set a breakpoint to suspend program execution.



Element	Description
command	A keyword-based Debugger command to execute when the breakpoint is reached. Search, Escape, and control-key commands are not valid here.
condition	A Boolean expression; this breakpoint suspends program execution when condition evaluates as TRUE (= any value except zero, FALSE, or NULL).
count	A negative integer. Execution is not suspended until this breakpoint is reached count times.
function	The identifier of a function. Between parentheses, this overrides the current function as the scope of any variable on which the breakpoint is set. Without parentheses, this breakpoint is set when function is called.
line	The line number in the current module (if no module is specified) at which this breakpoint suspends execution.
module	The filename of the source module at which this breakpoint suspends execution when program execution reaches the specified line.
name	The identifier that you declare for this breakpoint. The name specification must be enclosed between a pair of single (') or double (") quotation marks.

A breakpoint stops execution immediately prior to executing the 4GL statement specified by the BREAK command.

If you specify (*function*), the function that you name overrides the current function in the Source window in determining the scope of reference of any variable in a *variable* or IF *condition* specification.

The *name* option allows you to assign an optional name to a breakpoint. The *name* must start with a letter, and it must appear in the command line within single or double quotes.

If you specify a *count*, a minus symbol (-) must prefix it. After program execution reaches an enabled breakpoint *count* times, execution stops whenever that breakpoint is reached. That is, the Debugger does not wait another *count* times before stopping again but stops every time thereafter. The default value of *count* is `1`.

After a breakpoint stops execution, you can use CONTINUE or STEP to resume execution without resetting the *count*.

If you invoke a RUN command after a breakpoint stops execution, the Debugger resets *count* at your starting value and restarts execution.

If a (*function*) name is specified and *module* is omitted, then the *line* refers to the line number in the module that contains the function. If both the (*function*) and *module* names are omitted, then *line* refers to the module displayed in the Source window.

If the line is not an executable statement, the Debugger places the breakpoint at the start of the next executable 4GL statement, or at the last statement in the function (whichever is first). Variable definition statements, blank lines, and comments are not executable statements. If the *line* is not the first line of a 4GL statement, the breakpoint stops execution at the first line of the statement.

If the *module* contains no executable statements, an error message appears.

A *condition* cannot contain function calls.

If you specify both a *count* and an IF *condition*, *count* is reduced only if the *condition* is `TRUE`.

When you specify *function*, execution stops whenever the function is entered.

If you specify both *count* and *function*, the breakpoint is reached *count* times before execution stops at the *function* entry point.

You can specify the *line* of a 4GL statement that calls a C function or 4GL library function, but you cannot specify a *function* that is not a programmer-defined 4GL function.

You cannot abbreviate the keyword IF.

The IF condition can include the CURRENT and EXTEND() functions, as well as DATETIME and INTERVAL literals (including those produced by field-typing numbers using UNITS).

You can qualify variable names. (See "Scope of Reference" on page 9-16 for further information.)

There is no restriction on the number of breakpoints that you can set at any time.

If many breakpoints are simultaneously active, you might want to include distinctive PRINT messages as *command* options, to identify which breakpoint stops execution.

The scope of reference of any variables in the *commands* list is determined by their qualifiers and by the function that is current when the breakpoint takes effect.

The Debugger disregards whatever function was current when you issued a BREAK command, and ignores the (*function*) option, in identifying the scope of any variables in *commands*.

If several BREAK IF commands reference variables, the Debugger suspends execution if any of the variables changes when one of the breakpoint conditions is TRUE. You might need to specify DISABLE commands in the *commands* list to avoid unplanned breaks in this situation.

The Debugger assigns to each breakpoint a unique reference number. When you establish a new breakpoint, the Command window displays its reference number and other specifications. These can include the name, line number, function, module, IF conditions, commands to execute, and scope, depending on the arguments of your BREAK command.

Setting a breakpoint with the BREAK command also enables the breakpoint unless you include the asterisk (*). You can use the asterisk option to create a breakpoint without enabling it. The breakpoint is specified but disabled, as if you had set it and then used a DISABLE command to deactivate it.

Examples

Enter the following command to set an unnamed breakpoint at the tenth line of the current module:

```
break 10
```

The Command window displays its reference number, the word *break*, and its function, line number, module, and scope of reference.

The following command sets a breakpoint at line 100 in module **main**:

```
break main.100
```

If you enter the command:

```
break x
```

the Debugger first searches for a function named **x**. If **x** is a function in the current program, this breakpoint suspends program execution after a statement that calls **x**.

If no function named **x** is found, the Debugger searches for a variable called **x**. If variable **x** is found, the Command window displays the reference number of the new breakpoint, the word *break*, and the name of the variable. The next line displays its scope of reference. Program execution stops whenever the value of **x** changes.

If both a function and a variable have the name **x**, you must prefix the variable with appropriate qualifiers in a BREAK command. If neither a function nor a variable called **x** is found, the Command window displays the error message:

```
-16351: Variable [x] could not be located.
```

The command:

```
break x[i]
```

immediately evaluates array member **x[i]** and returns an error if **i** does not have a value. The command:

```
break if x[i]
```

treats **i** as a variable, and evaluates **x[i]** whenever the value of **i** changes during program execution.

The command:

```
break global.flag
```

stops execution whenever the value of the global variable **flag** changes. In the next example:

```
break if global.flag
```

program execution stops when **global.flag** becomes TRUE (nonzero), not whenever **global.flag** changes.

The command:

```
break "x" -2 20 if global.flag = 3 { print "Flag=3" }
```

sets a breakpoint named **x** at line 20 in the module currently displayed in the Source window. The breakpoint causes execution to stop if the breakpoint is reached twice, and the global variable **flag** equals 3 each time. When execution stops, the PRINT command is executed.

The command:

```
break (funcb) if a + b > 9
```

specifies that the scope of reference of variables **a** and **b** is the function called **funcb**, rather than the current function. Execution is suspended when the sum of these variables is greater than 9. The (*function*) specification always requires fewer keystrokes than when you use qualifiers, as in the equivalent command:

```
break if function.funcb.a + function.funcb.b > 9
```

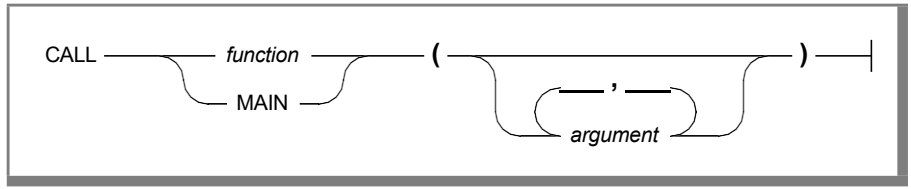
Related Commands

DISABLE, ENABLE, LIST, NOBREAK, STEP, TURN, WRITE

The data types supported by 4GL and the Debugger depend on which Informix database server you are using. See the *Informix Guide to SQL: Reference* for a list of supported data types.

CALL

Use CALL to execute a function and display returned values. The Application screen displays any output. Returned values appear in the Command window.



Element	Description
argument	The value of an argument to be passed to function.
function	The identifier of the function that you are invoking.

The function can be a programmer-defined 4GL function, a 4GL library function, a C function, or an ESQL/C function. See Appendix B for information about using the CALL command with 4GL programs that call C functions or ESQL/C functions.

The function can contain breakpoints and tracepoints. It does not need to be part of a completed 4GL program with a **main** program block.

You can CALL a function before a RUN command has started program execution. If a function includes SQL statements referencing a database that is not the current database, you must specify a database before you can CALL that function.

You can specify a database by a DATABASE command, or by a RUN command to start the current program, if the program includes an appropriate DATABASE statement.

It might be necessary to use the CLEANUP command to reinitialize variables and to close forms and windows before you reexecute a function by repeating a CALL command.

Any 4GL program variables that you specify as arguments of a CALL command must be active.

Examples

The following command executes the **main** section:

```
call main()
```

The next command executes a function called **find_cust** with arguments of 50 and 0:

```
call find_cust(50,0)
```

You can use CALL to execute a built-in 4GL function. For example, the command:

```
call  err_get(-408)
```

displays 4GL error message -408 in the Command window.

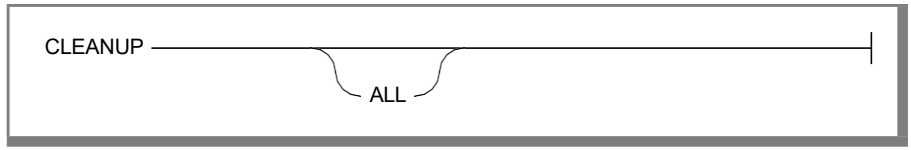
Related Commands

CLEANUP, DATABASE, FUNCTIONS, RUN, STEP, TRACE, VARIABLE, WHERE

The data types supported by 4GL and the Debugger depend on which Informix database server you are using. See the *Informix Guide to SQL: Reference* for a list of supported data types.

CLEANUP

Use CLEANUP to initialize all variables, to close all open windows and forms, and optionally to close the current database.The CLEANUP command is helpful when the function that you are executing with a CALL command has not reached a normal termination, but you want to CALL it again. In some debugging situations, this command is necessary before you can use a CALL command to restart a function that terminated abnormally.



If you enter the CLEANUP keyword with no option, the Debugger reinitializes all program variables and closes all windows and forms. The database remains open.

If you include the ALL option after the CLEANUP keyword, the Debugger reinitializes all the program variables, closes all windows and forms of the 4GL program, and closes the database.

The Debugger automatically closes the current database if the 4GL program stops because of a fatal error.

It is unnecessary to invoke a CLEANUP command before a RUN command. Invoking a RUN command automatically reinitializes all program variables and closes all forms and windows before execution starts.

You cannot use a CONTINUE or STEP command after a CLEANUP command until a RUN or CALL command resumes program execution.

Examples

In the sequence of commands that follows, the CLEANUP command closes all windows and forms and reinitializes all program variables but leaves the current database open:

```
call home(30,3)
...
cleanup
...
call home(30,3)
```

Without the CLEANUP command, open windows or forms or variables with unexpected values might interfere with the second attempt to call the function.

The command:

```
cleanup all
```

closes any open windows or forms, reinitializes all program variables, and closes the current database.

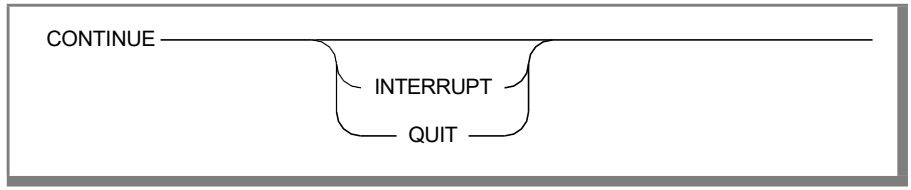
Related Commands

CALL, DATABASE, EXIT, LET, RUN

The data types supported by 4GL and the Debugger depend on which Informix database server you are using. See the *Informix Guide to SQL: Reference* for a list of supported data types.

CONTINUE

Use CONTINUE to resume execution of a program or to send an Interrupt or Quit signal to a currently running 4GL program.



CONTINUE without any options resumes execution at the first 4GL statement after an interrupt or breakpoint.

An error message appears if execution has not begun or has terminated, or if CLEANUP followed the last RUN or CALL command.

The Debugger traps all interrupts. The INTERRUPT and QUIT options send an Interrupt or Quit signal to your 4GL application to test its signal-handling code. Whether or not the INTERRUPT or QUIT option interrupts or terminates execution depends on how the specific 4GL program handles these signals.

Refer to the UNIX **stty** command for information about setting interrupt characters. On some systems the default is `CTRL-C`, but others use `DEL` or `BREAK`. `CTRL-\` is the default **quit** key on many UNIX systems. On systems that cannot generate a Quit signal, the QUIT option has no effect.

If the INTERRUPT or QUIT option terminates program execution, the Command window displays the name of the module, function, and line number of the current 4GL statement when execution stopped. Neither STEP nor CONTINUE can resume execution, but DUMP, PRINT, and WHERE can examine active variables and functions.

Examples

If program execution is suspended, the following command resumes execution of the current 4GL program or function:

```
continue
```

The next command sends an Interrupt signal to the 4GL application:

```
continue  int
```

The next command sends a Quit signal:

```
continue  quit
```

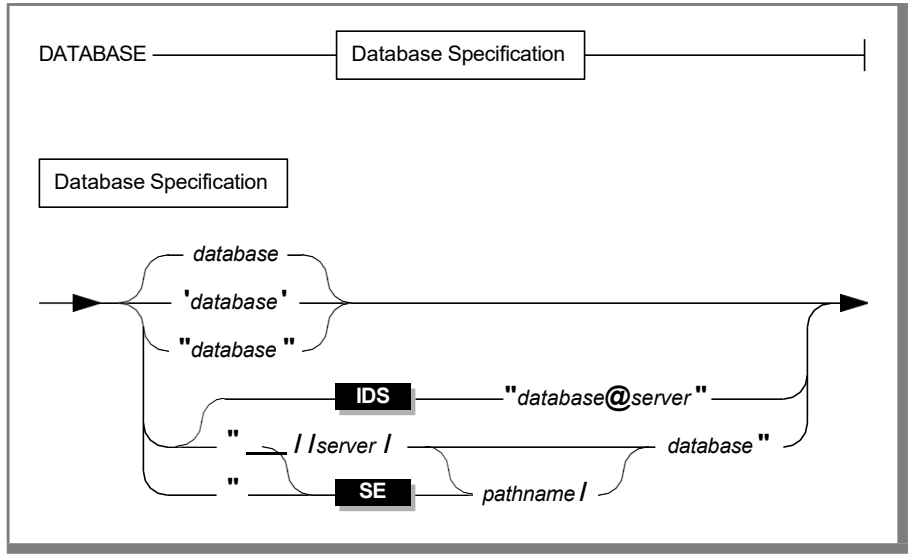
Whether this stops execution of the application depends on whether a DEFER QUIT statement in the 4GL code prevents the signal from passing through to the operating system.

Related Commands

BREAK, CALL, CLEANUP, INTERRUPT, STEP, RUN

DATABASE

Use DATABASE to specify the current database.



Element	Description
database	SQL identifier of an Informix database to which you seek a connection. (Blank spaces are not valid between quotes nor after the @ symbol.)
pathname	Path to the parent directory of the database (pathname/***database*.dbs**).
server	The network identifier of the host system where database resides.

If no database has been specified, a DATABASE command selects one. If you have a different current database, a DATABASE command closes it and replaces it with *database*.

If *database* is not in your current directory or in your **DBPATH** variable, then you must prefix it with a complete pathname.

Invoking the RUN command reopens the database specified in your program. To use a different database, you must set a breakpoint at the first line in the **main** program block and then invoke a DATABASE command to select a new database when that breakpoint is encountered. You can use CONTINUE to resume program execution.

Examples

The following command selects the **stores7** database:

```
database  stores7
```

The following sequence of commands sets a breakpoint at the beginning of **main**, begins running the program, selects a new database called **stereos**, and resumes program execution using the new database:

```
break main
run 9 5
database /u/dbtest/stereos
continue
```

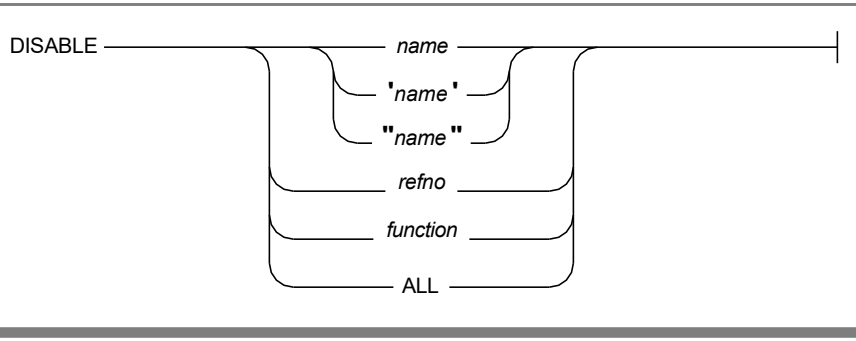
This example does not show an actual Command window display, which would also show output from several of these commands.

Related Commands

CALL, CLEANUP, RUN

DISABLE

Use DISABLE to turn off a breakpoint or tracepoint.



Element	Description
function	The identifier of a function, all of whose breakpoints and tracepoints are to be disabled.
name	The identifier of a breakpoint or tracepoint that is to be disabled.
refno	Integer code of a breakpoint or tracepoint that is to be disabled. (This reference number was assigned by the Debugger when a BREAK or TRACE command defined the specified breakpoint or tracepoint.)

The ALL option disables all breakpoints and tracepoints in the current 4GL program.

If a breakpoint or tracepoint is enabled and has the same *name* as *function*, only the breakpoint or tracepoint called *name* is disabled. If that breakpoint or tracepoint is already disabled, all breakpoints and tracepoints within *function* are disabled.

You have the option of enclosing the name of a breakpoint or tracepoint in single or double quotes.

DISABLE does not remove breakpoints or tracepoints. You can reactivate a disabled breakpoint or tracepoint at a later time by invoking the ENABLE command.

If a breakpoint or tracepoint named *all* is enabled, the DISABLE ALL command disables the breakpoint or tracepoint rather than disabling all of them.

Examples

The following command disables the breakpoint or tracepoint whose name is **kn**:

```
disable "kn"
```

An error message appears if **kn** cannot be located or if it is not active.

The command:

```
disable x
```

disables a breakpoint or tracepoint with name **x** or all breakpoints and tracepoints within the function **x**, while:

```
disable 1
```

disables the breakpoint or tracepoint with a reference number of 1.

The following command deactivates all of the breakpoints in the function **add_cust**:

```
disable add_cust
```

Related Commands

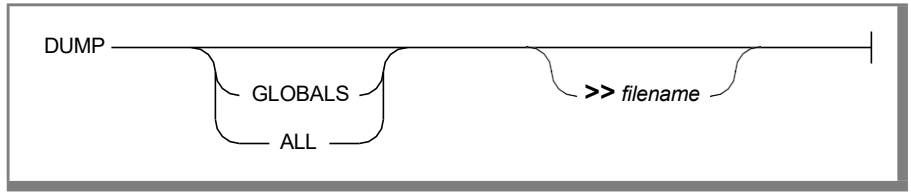
BREAK, ENABLE, LIST, NOBREAK, NOTRACE, TRACE, WRITE

DUMP

Use DUMP to write the names and values of variables in the current (most recently executed) function to the Command window or to an ASCII file.

An error message appears if you enter a DUMP command when no 4GL function is active. You cannot use DUMP to evaluate variables that exist only in C or ESQL/C functions.

After a fatal error or some other action causes the 4GL program to terminate abnormally, DUMP can evaluate global variables, and variables from the function in which execution stopped.



Element	Description
filename	The name of a file in which to write output from this command. This file specification can also include a pathname.

If you do not specify GLOBALS or ALL, only the variables in the current function are evaluated. If you specify GLOBALS, only global variables are evaluated. If you specify ALL, all global variables and all variables in the current function are evaluated.

If a file called *filename* already exists, the Debugger appends the names and values to the file. Otherwise, the Debugger creates the file. You can specify a pathname if you want the file saved outside your current directory.

Examples

The following command displays in the Command window the names and values of all the variables in the current function:

```
dump
```

The following command saves in a file called **fnstatus** the names and values of all global variables and of all the variables in the current function:

```
dump all >> fnstatus
```

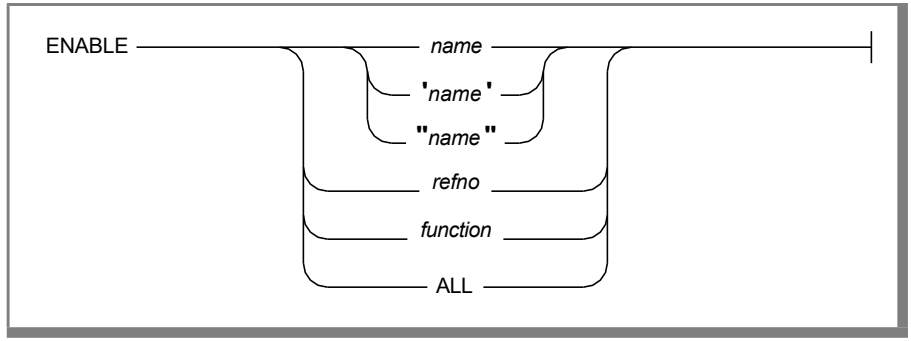
Related Commands

LET, TRACE, VARIABLE

The data types supported by 4GL and the Debugger depend on which Informix database server you are using. See the *Informix Guide to SQL: Reference* for a list of supported data types.

ENABLE

Use ENABLE to activate a breakpoint or tracepoint.



Element	Description
function	The identifier of a function, all of whose breakpoints and tracepoints are to be enabled.
name	The identifier of a breakpoint or tracepoint that is to be enabled.
refno	Integer code of a breakpoint or tracepoint that is to be enabled. (This reference number was assigned by the Debugger when a BREAK or TRACE command defined the specified breakpoint or tracepoint.)

The ALL option enables all breakpoints and tracepoints in the program.

If a disabled breakpoint or tracepoint *name* is the same as a *function* name, only the breakpoint or tracepoint *name* is enabled. If that breakpoint or tracepoint is already enabled, all breakpoints and tracepoints within *function* are enabled.

The *name* can be enclosed in single quotes or double quotes.

If a breakpoint or tracepoint called *all* exists, the ENABLE ALL command enables it rather than enabling all breakpoints and tracepoints.

Examples

The command:

```
enable x
```

enables the breakpoint or tracepoint called **x**, or enables all breakpoints and tracepoints within the function called **x**, if the current program calls a function of that name.

The following command enables the breakpoint or tracepoint with a reference number of 1:

```
enable 1
```

The command:

```
enable all
```

activates any disabled breakpoints or tracepoints in the program, or one called **all**, if a disabled breakpoint or tracepoint called **all** exists.

The following command activates a breakpoint or tracepoint called **me**:

```
enable "me"
```

Related Commands

BREAK, DISABLE, LIST, NOBREAK, NOTRACE, TRACE, WRITE

ESCAPE

Use the exclamation point (!) to send a command line to the operating system. You can use this feature to invoke an interactive program, such as a text editor, that requires keyboard input.

Element	Description
command	An operating system command to be executed during a debugging session. Press the exclamation key before entering command.

You must use the exclamation point (!) to invoke this command. The string `"_ESCAPE"` (with a leading underscore symbol) can be used as an argument of a HELP command.

When the command terminates, you are prompted to return to the Debugger. Pressing any key (except `!`) restores the Command and Source windows. The cursor returns to the window from which you escaped. If you type another exclamation point (!) when you are prompted to return to the Debugger screen, you can enter another operating system command line.

The ESCAPE key has nothing to do with the Escape feature of the Debugger.

Example

If you enter the following command at the Command window or Source window:

```
!csh
```

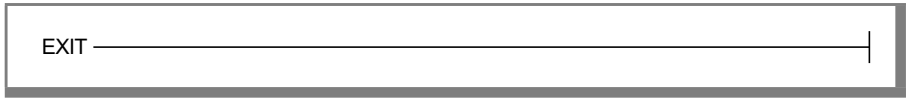
you create a new C shell. You can work at this shell and then return to the Debugger by entering `exit` (the system command, not the Debugger command) and then pressing any key.

Related Command

HELP

EXIT

Use EXIT to terminate a debugging session.



If you invoked the Debugger directly from the system prompt, EXIT returns you to that prompt. If you invoked the Debugger from the Programmer's Environment, enter `EXIT` to return to a menu.

If you are at the Application screen with a 4GL program waiting for input or producing output, you must first press the **Interrupt** key (typically CTRL-C) before you can exit.

Example

This command terminates the Debugger:

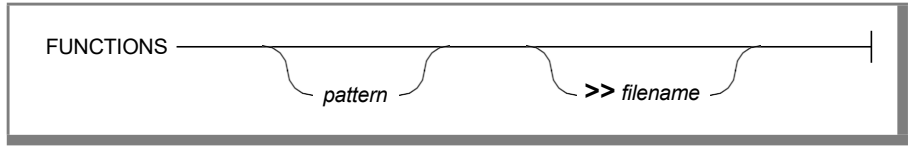
```
exit
```

Related Command

INTERRUPT

FUNCTIONS

Use FUNCTIONS to list in the Command window the names of programmer-defined 4GL functions in the current application. FUNCTIONS does not list the names of C or ESQL/C functions.



Element	Description
filename	The name of a file in which to write output from this command. This file specification can also include a pathname.
pattern	A string of literal characters and (optionally) special characters to use as a mask in searching for matching identifiers of functions.

If no search pattern specification follows the FUNCTIONS keyword, then the names of all functions in the current 4GL application program are displayed in the Command window.

The search pattern specifications can include a function name or any combination of characters, blanks, and the following special characters.

Character	Description
?	Match any single nonblank character
*	Match zero or more nonblank characters
[*d-q*]	Match any character between *d* and *q* (inclusive) in the ASCII collating sequence

If you specify a search pattern, FUNCTIONS displays every function name that matches the pattern.

If no function name matches a pattern, the Command window returns only the $ prompt. No additional message appears.

Examples

The command:

```
functions
```

returns the name of every programmer-defined function in the current 4GL program. The next command:

```
functions  add_cust
```

searches the source file for a function called **add_cust**. If the function is found, its name is displayed in the Command window. The next command:

```
functions  add*
```

specifies a wildcard pattern that matches the function in the previous example, as well as any other function whose name starts with **add**.

The following command lists the names of all the functions whose names contain the pattern **ust**, followed by any lowercase letter that comes after *m* and before *x*:

```
functions  *ust[n-w]*
```

Related Commands

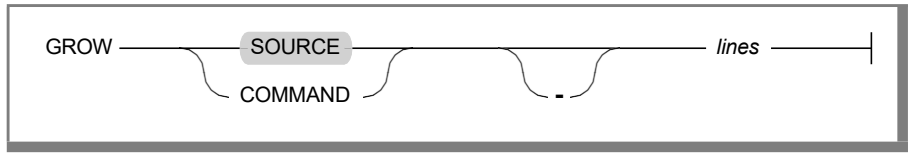
CALL, SEARCH, TRACE, VIEW, WHERE

GROW

Use GROW to change the size of the Command and Source windows. The GROW command changes the sizes of the Command and Source windows by modifying the values of the COMMAND LINES and SOURCE LINES parameters.

If you use a GROW command to change the sizes of the Source and Command windows, your new values persist until you change them with another GROW command or until an EXIT command ends the debugging session.

If you exit to the Programmer's Environment and then resume debugging the same 4GL program without returning to the operating system, your current window sizes are restored.



Element	Description
lines	The number of lines to add to the specified window.

The window that you specify increases by *lines* (or shrinks if *lines* is less than zero). The size of the other window changes reciprocally, so that on a standard 24-line display terminal, the sum of SOURCE LINES and COMMAND LINES is 19.

On a nonstandard terminal that can display *L* lines, the initial default size of COMMAND LINES is (*L*-4)/2. The initial default value of SOURCE LINES is one less than this.

The Source window line that displays the name of the 4GL source program module does not count as a line.

If *L* is the number of lines that your screen can display, the range of SOURCE LINES and of COMMAND LINES is from 1 to *(L*-6*)*. If a GROW command attempts to set a value of SOURCE LINES or COMMAND LINES outside of this range, then an error message appears, and the sizes of the windows are not changed.

The GROW command cannot change the size of the Application screen, the Help screen, or the operating system display.

Example

To increase the size of the Source window by five lines, enter any of the following equivalent commands:

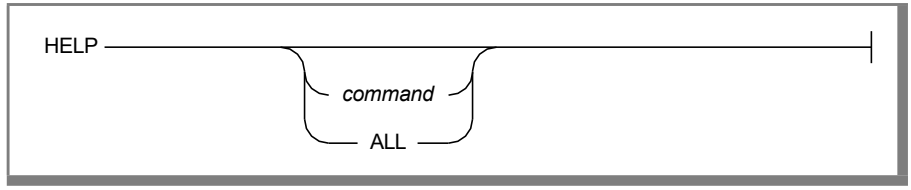
```
grow 5
grow source 5
grow command -5
```

Related Commands

LIST, WRITE

HELP

Use HELP for instructions on using Debugger commands.



Element	Description
command	The full or abbreviated name of any Debugger command.

If the command that follows the keyword HELP is the full or abbreviated name of a Debugger command, then the Help screen overwrites the Command window and Source window with information about how to use that command.

If no *command* argument is specified, then a menu-like list of HELP options, including every command keyword, appears. To display a message about any of these commands, select the appropriate keyword (by highlighting it with the arrow keys or by typing it and pressing RETURN).

The ALL option displays a brief synopsis of every command. The same synopsis appears if you enter a string after the HELP keyword that begins with a letter but does not correspond to a keyword. For example, your argument can be **v**, which is ambiguous, or **k**, which matches no command.

Besides the Debugger command keywords, the HELP command recognizes the options _ESCAPE, _INTERRUPT, _SCREEN, _SEARCH, _REDRAW, and _TOGGLE, representing Debugger commands that are controlled by special nonalphabetic characters. Each of these options is prefixed by an underscore (_).

If your argument after the HELP keyword begins with any other nonalphabetic character, an error message is displayed rather than a help message.

If a help message is longer than the current screen page, type S or press RETURN to see the next page.

After a help message has been displayed, type R or RETURN to restore the Debugger screen.

Examples

The following command displays a message about the TRACE command, based on the description of TRACE in this chapter:

```
help trace
```

Enter the following command to see the list of Help message topics:

```
h
```

You can use the arrow keys and press RETURN to display a message about a specific topic. If you enter:

```
help all
```

the screen displays a synopsis of Debugger command syntax.

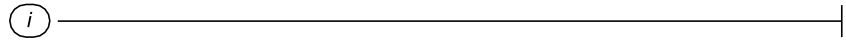
Related Commands

ESCAPE, INTERRUPT

INTERRUPT

Press the **Interrupt** key (typically CTRL-C) from the Application screen or the Source window to make the Command window your current window. If the 4GL program is running, pressing the **Interrupt** key suspends program execution at the current 4GL statement.

Like the names of the Escape feature and the Redraw, Screen, Search, and Toggle commands, INTERRUPT is not a Debugger keyword, but it appears at the top of this page and among the HELP topics so that you can find these notes. Prefix the word INTERRUPT with an underscore (_) when you use it as an option of the HELP command.



Element	Description
i	The **Interrupt** key, usually CTRL-C or DEL, or whatever physical key of the keyboard is assigned as the logical **Interrupt** key on your system.

The **Interrupt** key is the key specified by the **stty** command (often CTRL-C or DEL).

If the Source window is your current window when you press the **Interrupt** key, the cursor moves to the Command window and is ready for a command. (If EXITSOURCE is ON, pressing any alphabetic key has the same effect.)

If the Application screen is your current window, pressing the **Interrupt** key restores the Debugger screen. The Command window becomes your current window.

If a 4GL statement is executing, pressing the **Interrupt** key stops execution immediately after the statement. If DISPLAYSTOPS is ON, the next statement to be executed is highlighted in the Source window.

Pressing the **Interrupt** key stops execution of an application by interrupting the Debugger process that interprets the program rather than by passing an Interrupt signal to the 4GL program. You must use the CONTINUE INTERRUPT command rather than the **Interrupt** key if you want to test how a 4GL application handles an Interrupt signal.

Pressing the **Interrupt** key repeatedly while a 4GL program is running can produce unpredictable results. When the Application screen is your current window, for example, consecutive interrupts might result in an incorrect display of the Command and Source windows. If this happens, press CTRL-R to redraw the screen.

Example

If the Source window is your current window, pressing the **Interrupt** key makes the Command window your current window.

If you used the **Toggle** key to switch from the Source window to the Application window, pressing the **Interrupt** key switches to the Debugger screen and makes the Command window your current window.

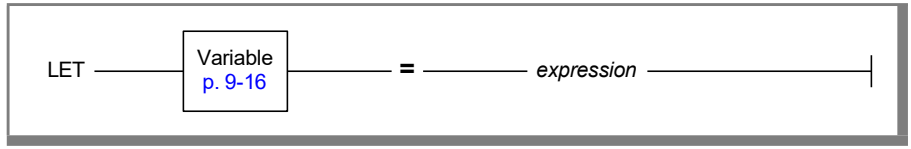
If the application program is waiting for input, pressing the **Interrupt** key suspends program execution after the current 4GL statement and makes the Command window your current window.

Related Commands

BREAK, CONTINUE, EXIT, REDRAW, TURN

LET

Use LET to assign an expression to a variable. If a 4GL program assigns to a variable a value that is different from what you intended, you can use the LET command to substitute the intended value. This allows you to continue examining a program without terminating a debugging session to modify and recompile the source code after you find the first flaw in program logic.



Element	Description
expression	An expression whose value is to be assigned to the variable.

Because a LET command has no effect on the source code, you must subsequently correct the source code if it assigns an improper value to a program variable.

The value that a LET statement assigns to a variable should be consistent with its data type declaration. If the two are different, the Debugger attempts to convert the value. This might result in truncation. An error message appears if the conversion fails.

The syntax of a LET command closely resembles the syntax of a 4GL LET statement. You must enclose character expressions in quotation marks.

An expression can contain a substring of a character array. It cannot include variables from multiple functions or variables from functions that are not active. You cannot use LET to reference variables unless program execution is suspended, or has terminated abnormally, after a RUN or CALL command.

The expression can include the following date and datetime functions.

DATE()	DAY()	MDY()	TODAY
MONTH()	WEEKDAY()	YEAR()	CURRENT

You can qualify the variable. See "Scope of Reference" on page 9-16 for more information.

If you follow a LET command with a RUN command, the RUN command will reinitialize all the program variables, restoring their original values.

To avoid having your work undone in this way, set a breakpoint before a 4GL statement that uses a variable whose value is incorrect. After the breakpoint stops program execution, use a LET command to assign a new value and invoke CONTINUE to resume execution with the corrected value.

Examples

The following command assigns the letter *y* to the variable **answer**:

```
let answer = "y"
```

The following command:

```
let global.rfrsh_flg = function.main.x/23
```

divides by 23 the variable called **x** that was defined in the **main** program block and assigns the resulting value to the global variable **rfrsh_flg**. If the data type of **global.rfrsh_flg** is INTEGER, the decimal portion of the value is discarded.

If **x** is of type CHARACTER, the command:

```
let x[101,101] = y
```

replaces the current value of the 101st character of **x** with the value of 4GL program variable **y**.

The following examples show the LET command with DATETIME and INTERVAL constants and with the CURRENT function:

```
LET dt1 = CURRENT

LET dt2 = DATETIME(1989-6-16 16:01:33) YEAR TO SECOND

LET dt3 = dt4

LET iv1 = INTERVAL(45-7) YEAR TO MONTH

LET iv2 = INTERVAL (12 15:25:35) DAY TO SECOND

LET iv3 = 5 UNITS DAY
```

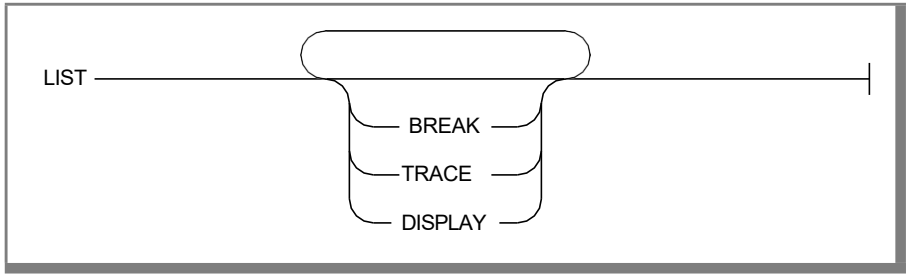
Related Commands

DUMP, PRINT, RUN, TRACE, VARIABLE

The data types supported by 4GL and the Debugger depend on which Informix database server you are using. See the *Informix Guide to SQL: Reference* for a list of supported data types.

LIST

Use LIST to display the current breakpoints, tracepoints, and terminal display parameters.



If you specify BREAK, the Command window displays the reference numbers and other specifications of the currently enabled and disabled breakpoints.

If you specify TRACE, the Command window displays the reference numbers and other specifications of the currently enabled and disabled tracepoints.

If you specify DISPLAY, the Command window lists the current values of the following display parameters:

- AUTOTOGGLE, when on, switches to the Application screen on output.
- DISPLAYSTOPS, when on, highlights in the Source window the next line to execute when the Debugger stops.
- EXITSOURCE, when on, switches from the Source window to the Command window at any alphabetic key.
- PRINTDELAY, when on, updates the Command window one line at a time or in blocks of several lines.
- SOURCETRACE, when on, highlights each line of source code as it executes.
- SOURCE LINES tells how many lines of source code appear in the Source window.
- COMMAND LINES tells how many lines of the command buffer appear in the Command window.
- TIMEDELAY SOURCE tells how long to pause between steps when SOURCETRACE is ON.

- TIMEDELAY COMMAND tells how long to pause between displaying successive lines of Debugger output in the Command window.
- APPLICATION DEVICE tells if a separate terminal has been specified for the display of the Application screen.

See the TURN command for more information about the parameters AUTOTOGGLE, DISPLAYSTOPS, EXITSOURCE, PRINTDELAY, and SOURCETRACE.

If you do not specify any option, the Command window displays all of the information described in this section.

Besides LIST, other commands that can display information about your current debugging session are ALIAS, DUMP, FUNCTIONS, PRINT, USE, VARIABLE, and WHERE.

Examples

The following command displays in the Command window all of the current breakpoints, tracepoints, and terminal display parameters:

```
list
```

The following LIST command displays all the breakpoints in the current 4GL program:

```
list break
```

This abbreviated command has the same effect:

```
li br
```

The following command lists the current display parameters and tracepoints:

```
list display trace
```

This abbreviated command has the same effect:

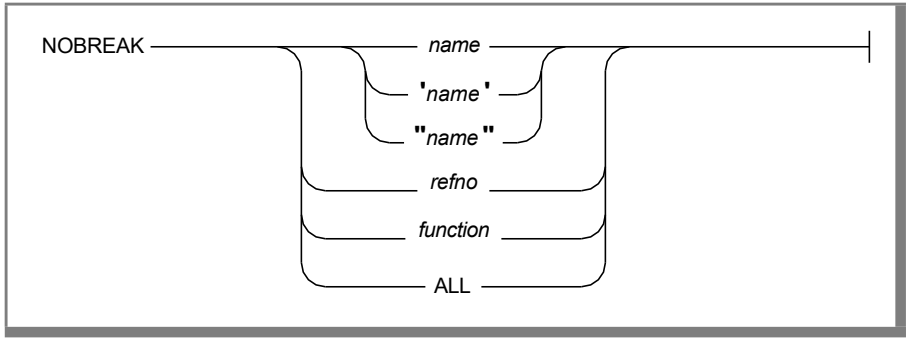
```
li d tr
```

Related Commands

APPLICATION DEVICE, BREAK, GROW, READ, TIMEDELAY, TRACE, TURN

NOBREAK

Use NOBREAK to remove a breakpoint. To deactivate a breakpoint that you might later want to reactivate, use the DISABLE command rather than NOBREAK.



Element	Description
function	The identifier of a function, all of whose breakpoints are to be removed.
name	The identifier of a breakpoint that is to be removed.
refno	Integer code of a breakpoint that is to be removed. (This reference number was assigned when a BREAK command defined the breakpoint.)

The LIST BREAK command displays the current reference numbers and names that can be used for the *refno* or *name* options of a NOBREAK command.

The ALL option removes all breakpoints in the program. If a breakpoint called **all** exists, the Debugger removes only that breakpoint rather than all of them.

If you specify *function*, the Debugger removes all breakpoints within the function. If a breakpoint and a function have the same name, only the breakpoint called *name* is removed. In that case, all breakpoints within *function* remain.

As an option, you can place *name* within a pair of single or double quotes.

Examples

The command:

```
nobreak  fnbr
```

removes a breakpoint called **fnbr** if a breakpoint of that name exists. Otherwise it removes all of the breakpoints in a function called **fnbr**. An unambiguous command to remove a breakpoint called **fnbr** is as follows:

```
nobreak  "fnbr"
```

The command:

```
nobreak  4
```

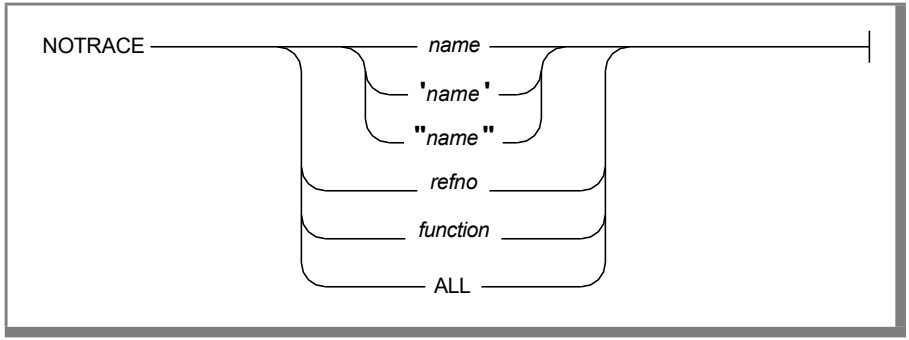
removes the breakpoint whose reference number is *4*.

Related Commands

BREAK, DISABLE, ENABLE, LIST, STEP, WRITE

NOTRACE

Use NOTRACE to remove a tracepoint. To deactivate a tracepoint that you might later want to reactivate, use the DISABLE command rather than NOTRACE.



Element	Description
function	The identifier of a function, all of whose tracepoints are to be removed.
name	The identifier of a tracepoint that is to be removed.
refno	Integer code of a tracepoint that is to be removed. (This reference number was assigned when a TRACE command defined the tracepoint.)

The LIST TRACE command displays the current reference numbers and any names that can be used for the *refno* or *name* options of a NOTRACE command.

The ALL option removes all tracepoints in the program. If a tracepoint called **all** exists, the Debugger removes only that tracepoint rather than all of them.

If you specify a function, the Debugger removes all tracepoints within the function. If a tracepoint and a function have the same name, only the tracepoint called *name* is removed. In that case, all tracepoints within *function* remain.

As an option, you can place *name* within a pair of single or double quotes.

Examples

The command:

```
notrace  fntr
```

removes a tracepoint called **fntr** if a tracepoint of that name exists. Otherwise it removes all of the tracepoints in a function called **fntr**. An unambiguous command to remove a tracepoint called **fntr** is as follows:

```
notrace  "fntr"
```

The command:

```
notrace  5
```

removes the tracepoint whose reference number is 5.

The next command:

```
notrace  all
```

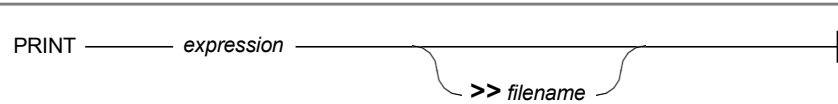
removes all tracepoints from the current program if no tracepoint with the name **all** exists.

Related Commands

DISABLE, ENABLE, LIST, TRACE, WRITE

PRINT

Use PRINT to display the value of an expression or save it in a file. You can use PRINT to display the current values of an entire record or array by specifying only its name as the expression.



Element	Description
expression	An expression whose value is to be displayed by this command.
filename	The name of a file in which to write output from this command. This file specification can also include a pathname.

An expression can be the name of a variable, a record, a date or datetime function, or an array. It can be a quoted string or an arithmetic expression. It cannot include nonglobal variables from multiple functions or from a function that is not active.

You can qualify variable names in an expression. If a nonglobal variable is defined in another active function that is not the current function, you must qualify the variable if you reference it in a PRINT expression. See "Scope of Reference" on page 9-16 for more information.

The expression can include the following date and datetime functions.

DATE()	DAY()	MDY()	TODAY
MONTH()	WEEKDAY()	YEAR()	CURRENT

The expression cannot reference any program variable unless a CALL or RUN command has started program execution.

An expression can specify substrings of a character array.

If you do not specify *filename*, the returned values are displayed in the Command window. The filename can include a pathname if you want the values saved in a disk file outside the current directory.

If you redirect output to a file that already exists, the Debugger appends the output to *filename*. Otherwise, the file is created.

You can use PRINT as a calculator during debugging sessions to perform arithmetic operations.

Examples

The following command displays the value of variable **x**:

```
print x
```

This can be a simple variable, a record, or an array. (You can use the abbreviated command **pr** instead of **print**.)

The following command divides 365 by 7 and displays the result in the Command window:

```
print 365/7
```

The following command displays an entire SQLCA record on the screen:

```
print sqlca
```

The following command displays the current system date and time:

```
print current
```

The following command copies to the disk file called **glbstatus** the current value of **sqlcode**:

```
print global.sqlca.sqlcode >> glbstatus
```

If **wrtschz** is the name of a variable of type CHARACTER, the command:

```
print wrtschz[1,20]
```

displays a substring that includes the first 20 characters of the current value of **wrtschz**.

Related Commands

CALL, DUMP, LET, RUN, TRACE, VARIABLE

The data types supported by 4GL and the Debugger depend on which Informix database server you are using. See the *Informix Guide to SQL: Reference* for a list of supported data types.

READ

The READ command enables you to execute Debugger commands that are specified in an ASCII file. These commands are executed, and their output is displayed in the Command window.

Element	Description
filename	The name of a file in which to write output from this command. This file specification can also include a pathname.

If you use a WRITE command to save features of a debugging session in *filename*, you can use a subsequent READ command to reestablish those features.

You can also create ***filename*.4db** with an editor, or you can use an editor to modify a **.4db** file that you created with WRITE, and then use READ to execute its commands.

The required extension of *filename* is **.4db**. You do not need to include the extension when you specify the filename in a READ command.

If you want to read a **.4db** file that is outside your current directory, you can prefix *filename* with a pathname.

You can include READ commands in the **.4db** file of a READ command. Error messages will appear, however, if you establish more than 10 levels of nested READ commands or if you attempt to READ a file that is still being processed by another READ command.

You cannot use READ to execute the following commands or control characters, which are not based on keywords.

Command	Function
Escape feature (!)	Operating system commands
Interrupt	Switches control to Command window
Search (/, ?)	Searches for patterns
CTRL-P	Saves screen display as a file
CTRL-Q	Enables terminal I/O
CTRL-R	Redraws screen display
CTRL-S	Suspends terminal I/O
CTRL-T	Toggles Application screen

Example

The following command reads and executes the Debugger commands contained in file **stdtest.4db**:

```
read stdtest
```

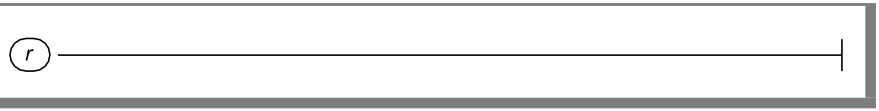
Related Commands

WRITE

REDRAW

Press CTRL-R to redraw the Source and Command windows, the Application screen, or Help screen. On some terminals, sequences of Debugger commands can sometimes produce an anomalous screen display. For example, repeatedly pressing the **Interrupt** key when the Application screen prompts you for input can produce unexpected results.

If you think that your Application screen, Help screen, Source window, or Command window has been incorrectly updated, press CTRL-R to redraw the current screen display.



Element	Description
r	The Redraw key, CTRL-R, which redraws the current window.

If your current window is the Command window or the Source window, pressing CTRL-R redraws both windows.

Like the names of the Escape feature, Interrupt, Screen, Search, and Toggle commands, REDRAW is not a Debugger keyword, but it appears at the top of this page and among the HELP topics so that you can find these notes. You must prefix the name REDRAW with an underscore(_) to use it as an option of the HELP command.

If you are using a second terminal to display the output of the application program, CTRL-R redraws both screens.

Example

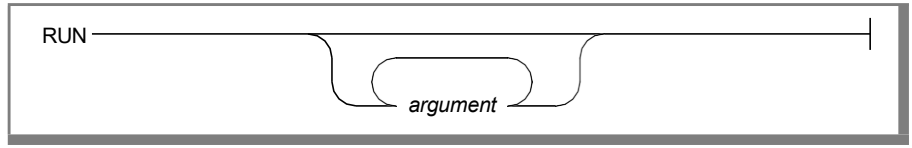
Pressing CTRL-R redraws the current screen.

Related Commands

APPLICATION DEVICE, HELP, INTERRUPT, SCREEN

RUN

Use RUN to start or restart execution of a 4GL program during a debugging session. Before a RUN command restarts a 4GL program, it first performs the functions of a CLEANUP command. The Debugger closes the database and any open windows or forms and initializes all program variables with zero or null values.



Element	Description
argument	The value of an argument to be passed to the program.

If multiple arguments are used, separate each one with spaces.

If you execute RUN more than once within the same debugging session, the same arguments are reused. You can change the arguments by specifying new ones when you execute RUN.

If you use RUN to restart a program that contains an enabled breakpoint with a *count* specification, RUN reinitializes *count* to the starting value. (Use CONTINUE or STEP if you want to maintain the current values.)

You cannot include RUN in the *commands* list of a tracepoint.

Example

The command:

```
run 1000 1
```

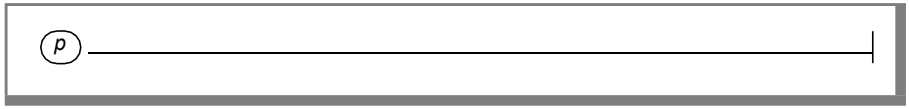
initializes all of the 4GL variables, closes any open forms or windows, and resets the counts of all breakpoints at their starting values. Then it starts (or restarts) the current 4GL application with two arguments.

Related Commands

BREAK, CALL, CONTINUE, INTERRUPT, STEP

SCREEN

Press CTRL-P to save the current display of the Source and Command windows or of the Application screen in a disk file. This facility enables you to record the displays of the Debugger and of the 4GL application program. Like other Debugger commands to display information, this can simplify the task of documenting debugging sessions.



Element	Description
p	*The **Screen** key,* CTRL-P, *which saves the current screen display in a file.*

Like the names of the Escape, Interrupt, Redraw, Search, and Toggle commands, SCREEN is not a Debugger keyword, but it appears here and among the HELP topics so that you can find these notes. You must prefix the name SCREEN with an underscore (_) to use it as an option of the HELP command.

If your current window is the Source window or the Command window, pressing CTRL-P saves the current screen display in a disk file called **fgldbscr**. If this file already exists, the Debugger appends the current display to the file.

If your current window is the Application screen, pressing CTRL-P saves the current screen display in a disk file called **fglapscr**. If this file already exists, the Debugger appends the current display to the file.

If you have used an APPLICATION DEVICE command to designate a second terminal for program output, your current window does not matter to the Screen command. After you press CTRL-P, the Debugger updates both the **fgldbscr** and **fglapscr** files with the current contents of the Debugger screen and of the Application screen, respectively.

Examples

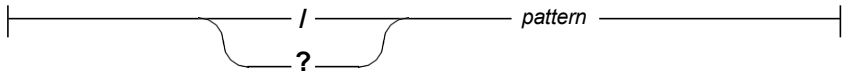
Pressing CTRL-P saves the current screen display in a file.

Related Command

APPLICATION DEVICE

SEARCH

Use Search commands to search for text patterns within the Source window or Command window. Do not confuse the Search command, which searches for patterns in text strings, with the USE command, which displays or specifies the directory search path for source files.



Element	Description
pattern	A string of literal characters and (optionally) special characters to use as a mask in searching for matching strings within text of the Command window or Source window (whichever is the current window).
/	The forward **Search** key.
?	The backward **Search** key.

You usually must use the slash (/) or the question mark (?) to initiate a search. The keyword SEARCH, prefixed by an underscore (_), is a HELP command option to display information about searching in the Source or Command window.

If you have already invoked a Search command from the Source window since the last VIEW command, pressing RETURN at that window repeats the last search from the current cursor position.

Searching forward means *down*, and backward means *up*, relative to the cursor position before a Search command.

The Debugger searches only within the current window. It searches the entire source module or command buffer in the specified direction, including patterns not currently displayed on the screen.

The Search command is case sensitive and does not regard any lowercase letter as matching any uppercase letter.

The previous pattern is considered the default if you specify no pattern.

You can use the following special characters.

Character	Description
?	Match any single nonblank character
*	Match zero or more nonblank characters
[*d-p*]	Match any letter between *d* and *p* (inclusive) in the ASCII collating sequence

A Search command does not require a leading and trailing asterisk (*) wildcard to find substrings. In this it differs from the FUNCTIONS command, which requires wildcard terminators to find embedded strings.

Examples

The command:

```
?a*b
```

searches backward in the current window for any string in which **a** precedes **b**. Here the question mark is the *Search backward* command key, not the wildcard.

The command:

```
/p?g
```

searches forward in the current window for any three characters beginning with **p** and ending with **g**. The command:

```
/[L-N]?a[a-b]d
```

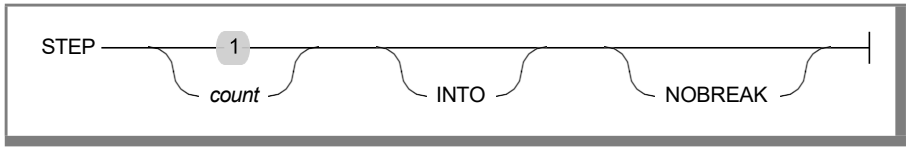
searches forward in the window for five-character strings, beginning with **L**, **M**, or **N**, followed by any character and ending in either **aad** or **abd**.

Related Commands

FUNCTIONS, VIEW

STEP

Use STEP to execute one or more individual 4GL statements. Each step is a single executable 4GL statement. Variable definition statements, blank lines, and comments are not executable statements.



Element	Description
count	The number of 4GL statements to be executed.

If the DISPLAYSTOPS display parameter is ON, the first step executes the statement that was highlighted in the Source window when you invoked the STEP command.

The default value of *count* is `1`. If *count* is not specified, only the next statement is executed.

A function call is treated as a single statement unless you specify the INTO option. If you specify INTO, each executable statement in a function is a step. You cannot STEP INTO a C function or a 4GL library function.

Execution stops if a step reaches a statement that contains an enabled breakpoint, unless you specify the NOBREAK option. If neither the INTO nor NOBREAK option is specified, execution stops if a step reaches a function that contains an enabled breakpoint.

If you specify NOBREAK, breakpoints have no effect while the STEP command is executing. This does not remove breakpoints, but it causes the Debugger to ignore breakpoints during the current STEP command.

Before you can invoke STEP, you must first begin execution with CALL or RUN, and then suspend execution with a breakpoint or with an Interrupt command. STEP elicits an error message after execution terminates or after a CLEANUP command.

Examples

The following command executes the next line as one statement, even if it includes a function call:

```
step
```

The following command executes the next 12 statements, not counting the statements inside functions as steps. Execution stops at enabled breakpoints or at the call of a function that contains an enabled breakpoint.

```
step 12
```

The following command executes the next 10 4GL statements. If a function call is encountered, executable statements within the function are counted as individual steps. If an enabled breakpoint is encountered before the 10 statements are executed, the program stops at the breakpoint.

```
step 10 into
```

The following command executes the next five statements, counting statements inside functions as steps. Execution does not stop at breakpoints.

```
step 5 into nobreak
```

Including or not including the INTO or NOBREAK options can produce very different results when you STEP through a 4GL program that calls a function containing a breakpoint. Suppose that your Source window displayed the following 4GL source code. An enabled breakpoint has been set at line 8, and program execution is currently suspended at line 2.

```
 1           MAIN
 2                CALL a()
 3           END MAIN
 4
 5           FUNCTION a()
 6                DEFINE i INTEGER
 7                LET i = 7
>8                DISPLAY i
 9                DISPLAY "DONE"
10           END FUNCTION
```

The table lists the STEP commands and the lines at which they stop executing.

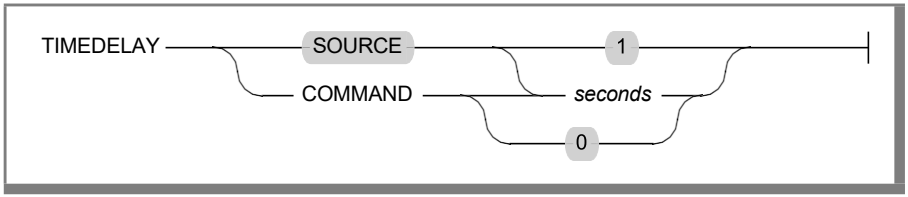
STEP Command	Stops at Line
STEP	8
STEP INTO	7
STEP NOBREAK	3
STEP INTO NOBREAK	7
STEP 3	8
STEP 3 INTO	8
STEP 3 INTO NOBREAK	9

Related Commands

BREAK, CALL, CONTINUE, ENABLE, DISABLE, FUNCTIONS, NOBREAK, RUN, TRACE

TIMEDELAY

Use TIMEDELAY to specify how quickly the 4GL application in the Source window executes when the SOURCETRACE display parameter is ON or how quickly successive lines of Debugger output appear in the Command window.



Element	Description
seconds	The duration of the delay, in seconds.

This command controls two independent display parameters, TIMEDELAY SOURCE and TIMEDELAY COMMAND. The first parameter affects the Source window, and the second affects the Command window.

The TIMEDELAY SOURCE command specifies how many seconds the current line in the Source window remains highlighted when SOURCETRACE is ON. The default window specification is SOURCE. The TIMEDELAY SOURCE command only affects Debugger operations when the SOURCETRACE parameter is ON.

The default value of TIMEDELAY SOURCE is `1`, corresponding to a one-second delay after each line of the 4GL application program in the Source window is highlighted.

A delay longer than the default value might be helpful when you are examining a brief or unfamiliar program, but it increases execution time by a factor of *seconds*.

To speed up a debugging session, you can set TIMEDELAY SOURCE to zero, the minimum value.

TIMEDELAY COMMAND specifies the number of seconds delay before successive lines of output from the Debugger appear in the Command window.

The default value of TIMEDELAY COMMAND is zero, corresponding to no delay between listing successive lines of Debugger output in the Command window. If output is being produced faster than you can read it, you can specify a delay of one or more seconds between new lines of output in the Command window.

On a fast system, a delay longer than the default value might be helpful when a command like LIST or READ is producing abundant output. An alternative to resetting TIMEDELAY COMMAND is to use the Search command or cursor movement keys to view lines of the command buffer that scrolled past too quickly to read.

Examples

The following command replaces the current value of TIMEDELAY SOURCE with 2:

```
timedelay  2
```

Because SOURCE is the default window specification, this is equivalent to the following command:

```
timedelay  source  2
```

When SOURCETRACE is ON, two seconds elapse between the execution of successive lines in the Source window.

The following command replaces the current value of TIMEDELAY COMMAND with 1:

```
timedelay  command  1
```

This causes the Command window to pause a full second between displaying successive lines of Debugger output.

Related Commands

LIST, TURN, WRITE

TOGGLE

Press CTRL-T to switch your screen display from the Debugger screen to the Application screen.

Element	Description
t	The **Toggle** key, CTRL-T, which switches the current screen display from the Debugger screen to the Application screen or vice versa.

If your current window is the Source window or the Command window, pressing the **Toggle** key (CTRL-T) displays the Application screen, so you can see the screen output of the 4GL application program.

You cannot supply input to the 4GL program after a Toggle command displays the Application screen. Pressing the **Toggle** key again restores the Debugger screen and the previous current window. Pressing any other key (except CTRL-S, CTRL-P, CTRL-Q, or CTRL-R) makes the Command window the current window.

Like the names of the Escape feature and the Interrupt, Screen, Search, and Redraw commands, TOGGLE is not a Debugger keyword, but it appears here and among the HELP topics so that you can find these notes. You must prefix the name TOGGLE with an underscore(_) to use it as an option of the HELP command.

If you have used an APPLICATION DEVICE command to display output from the 4GL application on a different terminal, then pressing CTRL-T has no effect.

Unless you use the **Toggle** key or an APPLICATION DEVICE command, the Application screen only appears when the 4GL program produces screen output or requires keyboard input. If you have entered a TURN OFF AUTOTOGGLE command, the Application screen does not appear when the program produces output unless you press CTRL-T.

If the 4GL program produces no output, the Application screen will be empty.

Example

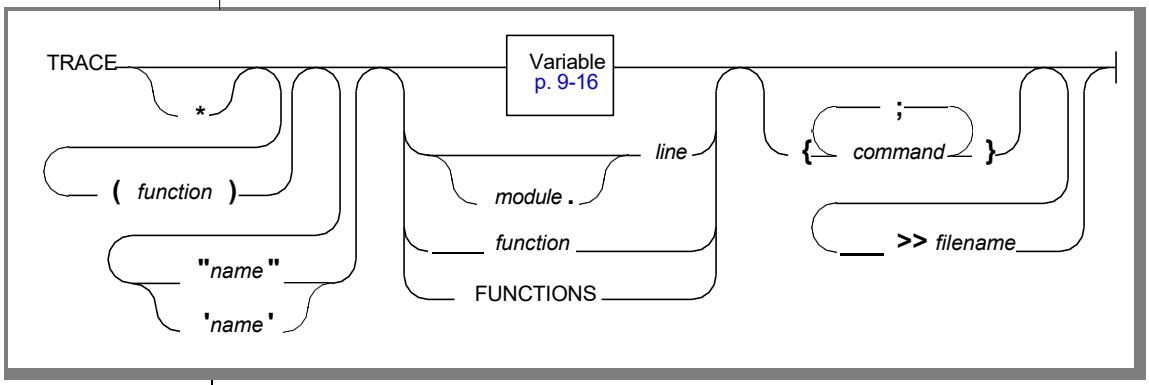
Pressing CTRL-T displays the Application screen.

Related Commands

APPLICATION DEVICE, INTERRUPT, SCREEN, TURN

TRACE

Use TRACE to show when a statement or function executes or when the value of a program variable changes. The Debugger assigns a unique reference number to each tracepoint. When the Debugger executes a TRACE command, the Command window displays the reference number and other specifications of the new tracepoint. These can include its name, line number, function, module, output file, commands to execute, and scope.



Element	Description
command	A keyword-based Debugger command to execute when the tracepoint is reached. Search, Escape, and control-key commands are not valid here.
filename	The name of a file in which to write output from this tracepoint. This file specification can also include a pathname.
function	The identifier of a function. Between parentheses, this overrides the current function as the scope of any variable on which the tracepoint is set. Without parentheses, this tracepoint is set when function is called.
line	The line number in the current module (if no module is specified) at which this tracepoint is set.
module	The filename of the source-code module at which this tracepoint is set when program execution reaches the specified line.
name	The identifier that you declare for this tracepoint. The name must be enclosed between a pair of single or double quotation marks .

If you include a (*function*) specification, the function that you name overrides the current function in the Source window in determining the scope of a variable on which you set a tracepoint.

If you specify a name, you must enclose the name in either single or double quotation marks. The name of a tracepoint must not duplicate the name of another breakpoint or tracepoint. It must start with a letter, but the subsequent characters can be letters, numbers, or underscores (_).

If the (*function*) name is specified and *module* is omitted, then *line* refers to the line number in the module that contains the function. If both the (*function*) and *module* names are omitted, then *line* refers to the module displayed in the Source window.

If *line* does not correspond to an executable statement, the tracepoint is set at the next executable statement following that line.

When you set a tracepoint on a variable, the variable name and its contents are displayed each time that the value of the variable changes. You cannot specify a variable that is a record or an array. You can, however, specify a variable that is a member of a record or an element of an array.

If you specify *commands* while tracing a function, the commands are executed only when the function is entered.

When you set a tracepoint on a function, the Debugger displays the function name and parameters when the function is entered. When execution of the function is completed, the Debugger displays the function name and returned values in the Command window. You can specify any function that the current program calls, including a C function, an ESQL/C function, or a 4GL library function.

If you specify a *commands* list, you must enclose it within braces ({ }). Use a semicolon to separate commands. The *commands* list cannot include a CALL, CONTINUE, STEP, or RUN command.

The scope of reference of any variables in the *commands* list is determined by their qualifiers and by the function that is current when the tracepoint takes effect. The Debugger disregards whatever function was current when you issued a TRACE command and ignores the (*function*) option in identifying the scope of any variables in *commands*.

You can qualify variable names. See the section "Scope of Reference" earlier in this chapter for more information.

The FUNCTIONS option sets a tracepoint on every function. You cannot abbreviate the FUNCTIONS keyword.

There is no restriction on the number of tracepoints that you can set at any time.

For typical debugging tasks, the PRINTDELAY parameter should be OFF when you are using tracepoints to trace program logic.

Setting a tracepoint with the TRACE command also enables the tracepoint, unless you include the asterisk (*) symbol. You can use the asterisk (*) option to create a tracepoint without enabling it. Use this option to define tracepoints that you would like to save for future debugging sessions, but that you do not want to be active in the current one. The tracepoint is specified but disabled, as if you had set it and then used a DISABLE command to disable it.

Examples

The following command establishes a tracepoint with no name at line 15 of the current module:

```
trace 15
```

The Command window displays its reference number, the word *trace*, and its function, line number, module, and scope of reference. For example:

```
(1)  trace main:15  [r_main.4gl]
          scope function: main
```

If you enter the command:

```
trace x
```

the Debugger first searches for a function called **x**. If **x** is a function of the current program, the Command window displays its name and parameters when the function is entered and its name and returned values after the function returns.

If no function called **x** is found, the Debugger searches for a variable called **x**, applying the scope of reference rules. If variable **x** is found, its name and value are displayed in the Command window whenever its value changes.

If both a function and a variable have the name **x**, you must prefix the variable with appropriate qualifiers in a TRACE command to set a tracepoint on the variable. If neither a function nor a variable called **x** is found, the Command window displays the following error message:

```
-16351: Variable [x] could not be located.
```

The following command displays the name and value of the **sqlcode** variable whenever it changes:

```
trace  global.sqlca.sqlcode
```

Because **sqlcode** is set at zero after each successfully executed SQL statement, this tracepoint identifies queries that return no rows, and it returns unsuccessfully executed statements.

The following command traces the execution of all functions and executes a PRINT command to display the current value of the global variable **sqlca.sqlcode** when each function is entered:

```
trace  functions  {print  global.sqlca.sqlcode}
```

The command:

```
trace  *  "vestige"  cmenu.12  {va  all}  >>  vestigial
```

specifies (but does not enable) a tracepoint called **vestige** at the 12th line of the module called **cmenu**. If you later enable this tracepoint and execute the 4GL statement in this line, the Debugger executes a VARIABLE ALL command when the tracepoint is reached and saves the output from this command in a disk file called **vestigial**.

The following command sets a tracepoint on member **b** of the record named **reca** in function **funca**:

```
trace  (funca)  reca.b
```

The Debugger will display the name and value of reca.b whenever it changes in **funca** but will ignore any variable called **reca.b** in other functions. The (*function*) specification requires fewer keystrokes than using qualifiers, as in the equivalent command:

```
trace  function.funca.reca.b
```

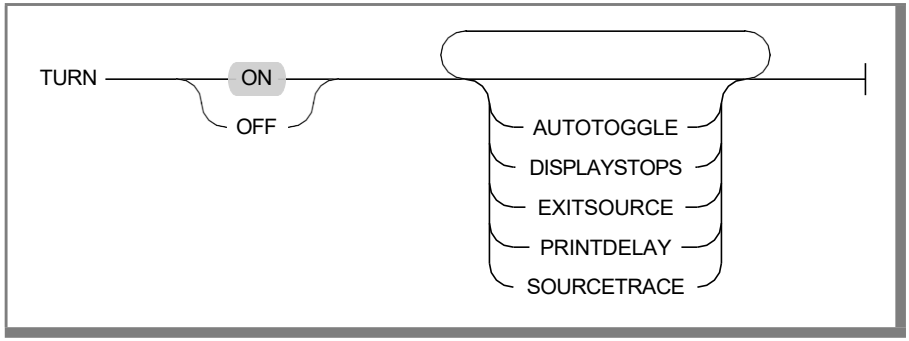
Related Commands

DISABLE, ENABLE, LIST, NOTRACE, TURN, WRITE

The data types supported by 4GL and the Debugger depend on which Informix database server you are using. See the *Informix Guide to SQL: Reference* for a list of supported data types.

TURN

The TURN command controls several terminal display parameters.



If you TURN OFF AUTOTOGGLE, there are only three situations in which you can see the Application screen:

- If keyboard input is requested by the 4GL program
- If you toggle to the display by pressing CTRL-T
- If you have directed output to a second terminal by an APPLICATION DEVICE command

If you TURN ON AUTOTOGGLE, the Application screen is displayed when the 4GL program produces screen output, as well as under the three preceding conditions.

If you TURN OFF DISPLAYSTOPS, after execution of a 4GL program stops, the Command window displays the next executable 4GL statement. The Source window does not display or highlight the next statement unless a fatal error occurs.

If you TURN ON DISPLAYSTOPS, after execution stops, the Source window displays and highlights the next statement that will be executed. The Command window displays information about the statement but not the actual statement.

If you TURN OFF EXITSOURCE, the only command that switches you from the Source to the Command window is an interrupt. If you TURN ON EXITSOURCE, then pressing any alphabetic key in the Source window makes the Command window your current window and echoes your keystroke after the $ prompt.

If you TURN OFF PRINTDELAY, the Command window displays output by scrolling up a single line at a time. If you TURN ON PRINTDELAY, the screen is updated in blocks of lines (about a third of the Command window) when a command produces multiple lines of screen output.

If you TURN OFF SOURCETRACE, the Source window does not highlight lines as they are executed, and does not adjust its display to include the currently executing line. (If an error occurs, however, the Source window always displays and highlights the line containing the error.) If you TURN ON SOURCETRACE, the Source window displays and highlights each line as it executes.

Unless a TURN command or a **.db4** file establishes other values, the default values of these display parameters are equivalent to the results of the following commands:

- TURN ON AUTOTOGGLE
- TURN ON DISPLAYSTOPS
- TURN ON EXITSOURCE
- TURN OFF PRINTDELAY
- TURN OFF SOURCETRACE

You can use the LIST DISPLAY command to show the current values of these parameters in the Command window.

The current values of these display parameters persist until another TURN command changes them or until you end the debugging session with an EXIT command. If you exit to the Programmer's Environment and then resume debugging the same 4GL program without returning to the operating system, your current display parameters are restored.

Examples

The command:

```
turn off autotoggle
```

prevents the Application screen from appearing when the current 4GL program produces output without requiring input.

The following command specifies that DISPLAYSTOPS and PRINTDELAY are ON:

```
turn displaystops printdelay
```

When DISPLAYSTOPS is ON, the Source window displays the current 4GL statement after program execution stops.

When PRINTDELAY is ON, multiple-line output from a Debugger command is added to the command buffer in blocks of lines, rather than a single line at a time.

The following command turns OFF the AUTOTOGGLE, EXITSOURCE, and SOURCETRACE parameters:

```
turn off a e s
```

It is usually convenient to abbreviate the options of TURN because if you misspell a keyword, your display parameters are not modified, and an error message appears. For example, an invalid TURN command such as:

```
turn on autototoggle
```

elicits the following error message:

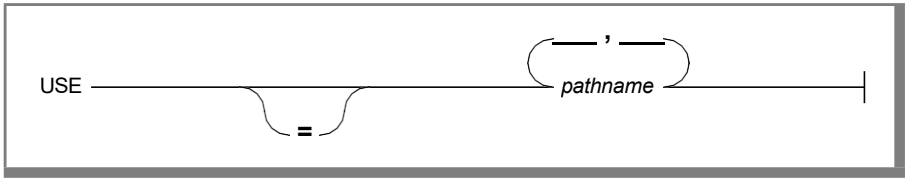
```
-16364: Unknown option [autototoggle]
```

Related Commands

INTERRUPT, LIST, WRITE

USE

Use the USE command to specify or display the source file search path. The USE command allows you to see the source code when you step into a module that is not in your current directory. It does not execute an alternate p-code. You still need to **fglpc** each module and **cat** them together to make the executable p-code.



Element	Description
pathname	The full pathname or relative pathname of a directory to be included in the search path of the **Debugger** for **.4gl** source files.

Enter a USE command without any arguments to display your current directory search path for 4GL source files.

If any of your source files are in directories outside your current search path, the USE command enables you to add the pathnames of one or more additional directories to the current directory search path. If you list multiple pathnames in a USE command, the first directory that will be searched is specified by the first pathname, the second by the second pathname, and so forth.

If you include the equal sign (=) in a USE command, the new pathnames replace the current search path. If you do not include the equal sign (=), the *pathnames* take precedence in the order of search, ahead of the following directories:

- Directories from previous USE commands
- The current directory
- The directory associated with the 4GL program
- Directories listed after the **-I** symbols in a **fgldb** command
- Directories specified in your **DBSRC** environment variable

A modified directory search order that you establish by a USE command only persists for the duration of the current debugging session or until your next USE command. If you exit to the Programmer's Environment and then resume debugging the same 4GL program without returning to the operating system, the Debugger restores any source file search path that was in effect when you ended the previous debugging session.

If you plan to reestablish the same search order for a subsequent debugging session, you can save the current search order with a WRITE DISPLAY command.

Examples

The following command displays your current search path in the Command window:

```
use
```

After the following USE command, the first directory in your current search path will be **/b/low**, and the second will be **/m/b/high**:

```
use  /b/low,  /m/b/high
```

After the second of the following two commands, the first directory in the current search path will be **/m/b/high**, and the second will be **/b/low**, because the most recent USE command takes precedence:

```
use  /b/low
  . . .
use  /m/b/high
```

The next command replaces the current source file search path with the current directory:

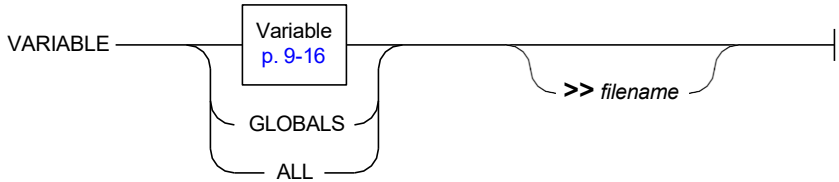
```
use = .
```

Related Command

WRITE

VARIABLE

Use VARIABLE to display the declaration of a variable, including the data type specification and the scope of reference.



Element	Description
filename	The name of a file in which to write output from this command. This file specification can also include a pathname.

If you do not specify *variable*, GLOBALS, or ALL, the type of each variable in the current function is displayed, but global variables outside the current function are ignored.

If the variable is a record, the name and data type of each component variable are displayed.

You can qualify the scope of reference of a variable by prefixing its name with a qualifier. Possible qualifiers include those in the following list.

Qualifier	Scope
GLOBAL	In all modules
MODULE.*mod-name.*	In the specified module
FUNCTION.func-name.	In the specified function
rec-name[.*rec-name* …].	In the record [within another record …]

If you specify GLOBALS, the Debugger only returns the types of the global variables.

If you specify ALL, the Debugger returns the data types of all global variables in the program and of all local variables in the current function.

The VARIABLE command can only return the declarations of variables in 4GL functions. If a C or ESQL/C function is the current function, only the ALL and GLOBALS options return data type information. Both options declare all the global variables but no local variables.

If you specify an output file, the display is redirected to the file. If the file already exists, the output is appended to it.

Examples

The following command returns the types of the local and global variables that appear in the current function:

```
variable
```

The next command displays declarations of all the global variables in the program:

```
variable  global
```

The following command:

```
variable   function.add_order.num_cust
```

displays the declaration of the variable **num_cost** in a function called **add_ord**.

If you enter the command:

```
variable  all  >>  snapshot
```

the Debugger saves the declarations of all global variables in the program and any other variables of the current function in a file called **snapshot**. This output is not displayed in the Command window.

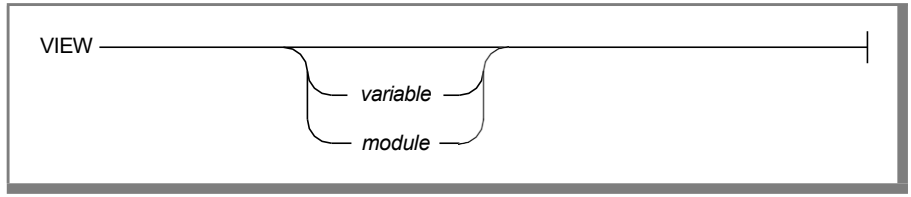
Related Commands

DUMP, LET, PRINT

The data types supported by 4GL and the Debugger depend on which Informix database server you are using. See the *Informix Guide to SQL: Reference* for a list of supported data types.

VIEW

Use VIEW to move the cursor to the Source window and optionally display a specific 4GL function or module. You can use cursor movement commands or Search commands to move the cursor or to scroll the current module within the Source window. Press the **Interrupt** key to return to the Command window.



Element	Description
function	The identifier of the function to be displayed.
module	The filename of the source-code module to be displayed.

If you enter the VIEW keyword without additional options, the same lines that were displayed in the Source window when you invoked the VIEW command remain in view.

The Debugger displays in the Source window whatever 4GL function or module you specify in a VIEW command. If necessary, the current module is replaced by the module that you specify or with the function that you specify.

Do not include an argument list with a function.

The VIEW command can only display a 4GL function or a 4GL module. You cannot specify a 4GL library function, a C function, or an ESQL/C function in a VIEW command.

If both a function and a module have the same name, the Source window displays the function.

Examples

The following command makes the Source window your current window:

```
view
```

The following command also makes the Source window the current window:

```
view add_cust
```

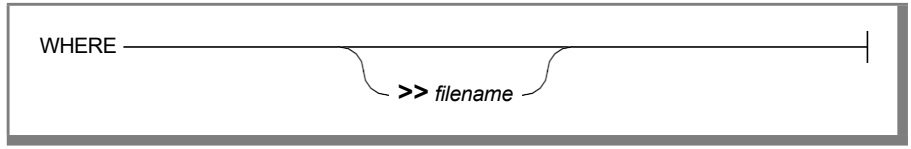
It displays the source code of the function **add_cust** or the module of the same name if no function exists.

Related Commands

CALL, FUNCTIONS, INTERRUPT, TURN

WHERE

Use WHERE to display the name and arguments of each 4GL function that was called to reach the current 4GL statement or to redirect this information to a file. WHERE displays the current list of active functions. These are functions that were called before the current 4GL statement was executed but that have not yet returned. Besides the names of these functions, WHERE displays the line and source module from which each function was called and evaluates any parameters that were passed with it.



Element	Description
filename	The name of a file in which to write output from this command. This file specification can also include a pathname.

You can specify a filename in which to save the output from a WHERE command. This can be outside your current directory if you prefix *filename* with a pathname.

If a file called *filename* already exists, the WHERE command appends the returned values without overwriting the file.

WHERE only describes programmer-defined 4GL functions. An exception occurs, however, if WHERE appears in the command list of a tracepoint that you set on a C function, on an ESQL/C function, or on a 4GL library function. If the tracepoint is reached, WHERE evaluates any parameters that were passed. It lists the function as a C or 4GL library function, rather than the module and line number of the calling statement, and lists the active functions.

An error message appears if you invoke the WHERE command when no 4GL program is currently executing. You can use WHERE when execution is suspended or after a program or function terminates abnormally.

Examples

The following command:

```
where
```

displays the programmer-defined functions that were executed to get to the current 4GL statement, as well as the module and line number from which each function was called and any parameters that it passed.

The next command saves the results of a WHERE display in a disk file called **myfile** in a subdirectory called **/m/tom**:

```
where >> /m/tom/myfile
```

If you have not yet issued a RUN or CALL command since you began the debugging session or since the last CLEANUP command, no function is active. If you use WHERE when there are no active functions, the following error message appears:

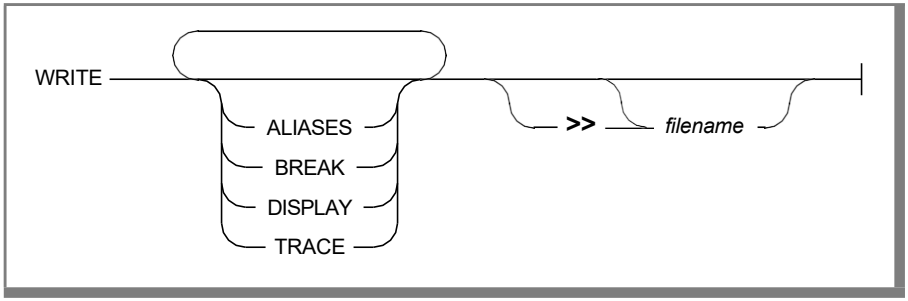
```
-16387: Program is not currently being executed.
```

Related Commands

CALL, CLEANUP, FUNCTIONS, RUN, TRACE

WRITE

Use WRITE to save in a file the commands to establish the breakpoints, tracepoints, aliases, directory search path specifications, or terminal display parameters. The resulting output file can be used as an initialization file or with a subsequent READ command to reestablish the current debugging session.



Element	Description
filename	The name of a file in which to write output from this command. This file specification can also include a pathname.

If no other keyword appears in a WRITE command, then by default *all* of the current breakpoints, tracepoints, terminal display parameters, aliases, and the source file search path are saved in the output file.

The DISPLAY option saves in the output file commands to establish the current values of your terminal display parameters and source file search path. In addition, a LIST DISPLAY command is appended to the output file. This displays the restored values in your Command window when the file is used as an initialization file or in a READ command.

If you include the **>>** symbols and specify a filename, the WRITE command assigns a name to the output file by appending the extension **.4db** to your filename.

If you do not include a filename specification, WRITE assigns a default name to the output file by appending the extension **.4db** to the filename of the **.4gi** or **.4go** program that you are currently debugging.

If a **.4db** file with the same name as the output file already exists in the same directory, information in the existing file is not overwritten. Instead, WRITE appends commands to the existing file.

If you want to save the output file in a directory other than your current directory, you must prefix *filename* with the complete pathname.

You can use an editor to supplement, delete, or modify the commands in an output file. The file cannot include nonkeyword commands such as Interrupt, Screen, or Toggle.

You can edit the output file of a WRITE command to include READ statements that specify other **.4db** input files. The Debugger will display an error message, however, if READ commands are nested more than 10 deep or if nested READ commands attempt to access each other.

Examples

This WRITE commands saves commands in the file **strtdbg.4db**:

```
write >> strtdbg
```

Because no restriction is included, the output file includes commands to establish all the current breakpoints, tracepoints, terminal display parameters, search paths, and aliases. The >> *filename* specification is unnecessary if you want to save the commands in a file whose filename is the same as that of the current program but with extension **.4db**.

The following command:

```
write display >> /k/leslie/myfile
```

saves commands to establish the current display parameters and search path (but no ALIAS, BREAK, or TRACE commands) in a file called **myfile.4db** in directory **/k/leslie**.

The >> symbols are usually optional unless the name of the output file could be confused with one of the options. These symbols could be omitted in the previous example but not in the command:

```
write a b >> d
```

which specifies that the ALIAS and BREAK commands will be saved in a file named **d.4db**. If you omit the redirect symbol in this example, the Debugger interprets **d** as the abbreviation of DISPLAY.

Related Commands

ALIAS, APPLICATION DEVICE, BREAK, GROW, LIST, READ, TIMEDELAY, TRACE, TURN, USE

Appendix

A

Environment Variables

Like INFORMIX-4GL, the Debugger makes the following assumptions about the user's environment:

- The editor used is the predominant editor for the operating system (usually **vi**).
- The desired database is in the current directory.
- If the computer is running UNIX System V, the program that sends files to the printer is usually **lp**. For other UNIX systems, the default is **lpr**.
- The **/tmp** directory stores temporary files.
- The 4GL and Debugger programs and their associated files are located in the **/usr/informix** directory.

You can change any of these assumptions by setting one or more environment variables.

Setting Environment Variables

You can set environment variables at the system prompt, in your **.profile** file (if you are using the Bourne shell), or in your **.cshrc** or **.login** file (if you are using the C shell).

If you set an environment variable at the system prompt, you will have to assign it again the next time you log on to the system. If you set a variable in your **.profile** file (Bourne shell) or in your **.cshrc** or **.login** file (C shell), UNIX will assign it automatically every time you log on to the system.

Use the following formats to set environment variables in the C and Bourne shells:

- If you are using the C shell, enter the following command to set the **ABCD** environment variable to *value*:

```
setenv ABCD value
```

- If you are using the Bourne shell, enter the following two commands to set the **ABCD** environment variable to *value*:

```
ABCD=value
export ABCD
```

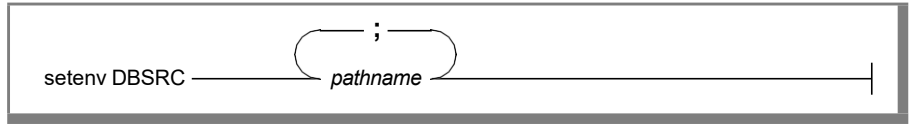
The environment variables recognized by the Debugger (and by 4GL, except for **DBSRC**) are as follows:

- **DBANSIWARN**
- **DBDATE**
- **DBDELIMITER**
- **DBEDIT**
- **DBLANG**
- **DBMONEY**
- **DBPATH**
- **DBPRINT**
- **DBSRC**
- **DBTEMP**
- **INFORMIXDIR**
- **SQLEXEC**

This appendix describes only the **DBSRC** environment variable. For information about the other environment variables, see the *INFORMIX-4GL Reference* manual.

DBSRC

This environment variable specifies directory pathnames that will be part of the search path only during debugging sessions.



Element	Description
pathname	is the full pathname of a directory for **.4gl** source files.

For example, if you specify:

```
DBSRC=/b/dawn/programs:/b/robin
```

then the Debugger will search for files in the **programs** and **robin** directories. You are not required to specify a **DBSRC** variable.

This is the only environment variable that is not used in the same way by the Debugger and by 4GL. When you exit from the Debugger and return to the Programmer's Environment or to the operating system, the pathnames specified in **DBSRC** are no longer part of the search path.

See "Specifying the Source Program Search Path" on page 8-19 for more information on specifying the order of directory search of the Debugger.

Appendix

B

Calling C Functions

INFORMIX-4GL is a powerful fourth-generation language whose commands support the database management requirements of typical business, accounting, administrative, and information retrieval applications. In some situations, however, programmers with experience in the C language might want to incorporate one or more C functions or INFORMIX-ESQL/C functions in a 4GL program. The following list gives examples of some of the tasks for which a C function might be appropriate:

- Displaying operating system environment information
- Making operating system calls
- Using library functions (such as mathematical functions)
- Interfacing with a nonstandard I/O device

The Debugger supports facilities for developing 4GL programs that include C functions. When calling a C function from a 4GL program, you must use 4GL popping and pushing functions in your C code when passing or returning values. See the *INFORMIX-4GL Reference* for more information about C functions.

This appendix describes how to use the Debugger to examine a 4GL program that calls C functions. These procedures require that you edit a C source file called **fgiusr.c**, that you compile your 4GL functions, and that you create a modified version of the Debugger. You can invoke this customized Debugger to examine your application in a debugging session.

Creating, Compiling, and Debugging

Figure B-1 shows the process of creating, compiling, and debugging a multi-module 4GL program from the command line. If you compare this diagram to Figure 8-3 on page 8-16, you will notice several differences between debugging an ordinary 4GL program and debugging one that calls C functions:

- You edit file **fgiusr.c** to declare your C functions.
- You invoke a program called **cfgldb**.
- You create a customized Debugger.
- You invoke the customized Debugger.

Here the rectangles represent specific commands, and the circles represent files. If your program includes only one **.4gl** source module, you will not require the **cat** utility to combine 4GL modules.

Figure B-1 does not show how to create a customized p-code runner. That process is described later in this appendix and requires that you substitute the command **cfglgo** for **cfgldb** in the procedure shown in Figure B-1. This produces a customized runner for your users to execute the 4GL program that calls C functions or ESQL/C functions.

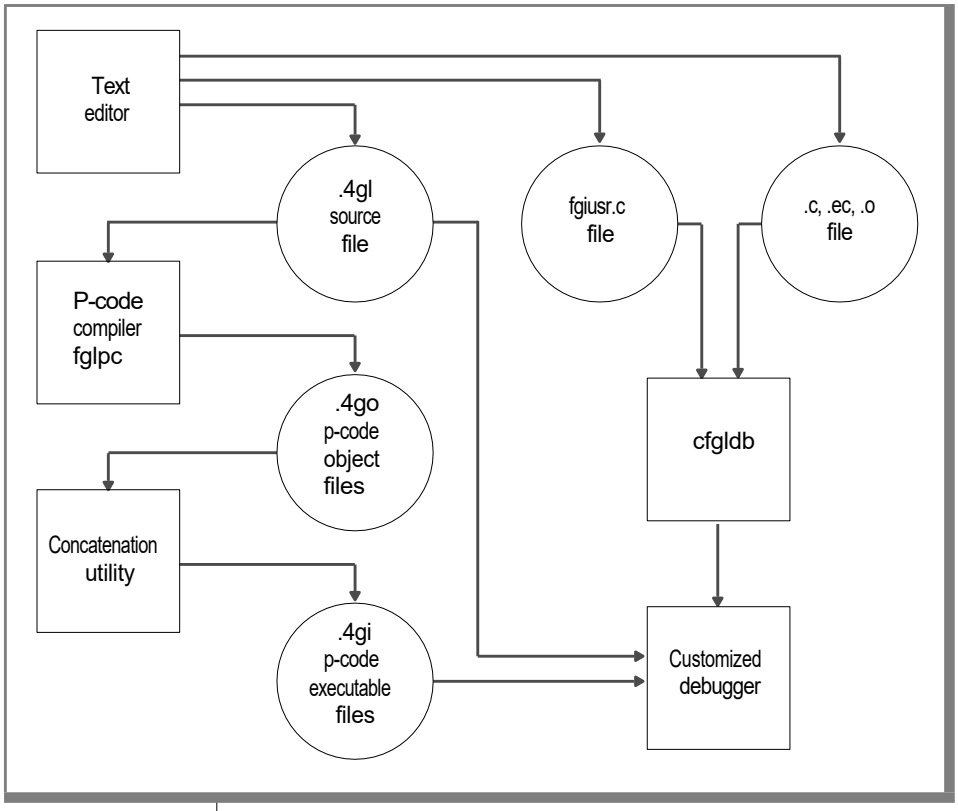


Figure B-1
Debugging a Program That Calls C Functions

The fgiujsr.c Structure Definition File

The **install** program that is distributed with the Debugger copies a file called **fgiusr.c** into directory **$INFORMIXDIR/etc**:

```
/*********************************************************
*                                                        *
*            INFORMIX  SOFTWARE,  INC.                   *
*                                                        *
* Title:  fgiusr.c                                       *
* Sccsid:  @(#)fgiusr.c     4.2    6/26/99   10:48:37     *
* Description:                                           *
*         definition  of  user  C  functions             *
*                                                        *
*********************************************************
*/

/*******************************************************
* This  table  is  for  user-defined  C  functions.
*
* Each  initializer  has  the  form:
*
*   "name",  name,  nargs
*
* Variable  #  of  arguments:
*
*   set  nargs  to  -(maximum  #  args)
*
* Be  sure  to  declare  name  before  the  table  and  to  leave  the
* line  of  0's  at  the  end  of  the  table.
*
* Example:
*
*   You  want  to  call  your  C  function  named  "mycfunc"  and  it
*   expects  2  arguments.   You  must  declare  it:
*
*        int  mycfunc();
*
*   and  then  insert  an  initializer  for  it  in  the  table:
*
*        "mycfunc",  mycfunc,  2
*******************************************************
*/

#include "fgicfunc.h"

cfunc_t  usrcfuncs[]  =
     {
     0,  0,  0
     };
```

The **fgiusr.c** file is a data file in which you must specify information about programmer-defined C or ESQL/C functions. You must edit this file to declare any C functions called from your 4GL program. Syntax of an edited **fgiusr.c** file follows:

```
[/* comments */]
#include "fgicfunc.h"
   int function1()[;
   . . .
   int functionN()]
cfunc_t usrcfuncs[] =
   {
   "function1", function1, [-]args1,
   . . .
   "functionN", functionN, [-]argsN,
   0, 0, 0
   };
```

In this example, *comments* are instructions that describe how to edit the **fgiusr.c** file. The *function1, ... , functionN* specifications are the names of one or more C functions and ESQL/C functions that are called from the 4GL program. The *args1, ... , argsN* specifications correspond to the argument numbers for each function.

The Debugger cannot be used with 4GL programs that call C functions unless you edit this file to declare each C function or ESQL/C function. When editing the **fgiusr.c** file, keep the following considerations in mind:

- If the number of arguments of a function can vary, enter the maximum number of arguments, prefixed by a minus sign (-).
- The **fgiusr.c** file can declare your functions in any order.

Example

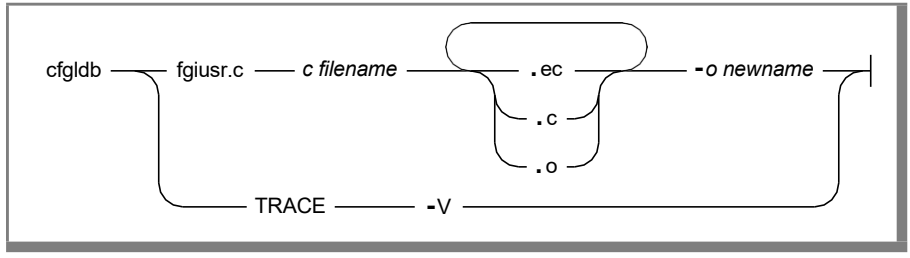
In the following example, four lines have been inserted into **fgiusr.c** to describe two C functions, **func1** and **func2**:

```
#include "fgicfunc.h"
int func1();
int  func2();
cfunc_t  usrcfuncs[]  =
     {
     "func1",  func1,  2,
     "func2",  func2,  -3,
     0,  0,  0
     };
```

The numbers in the lines before the three zeroes specify that **func1** requires two arguments and that **func2** can accept no more than three arguments.

Creating a Customized Debugger

The Debugger includes a facility by which the edited **fgiusr.c** file can be compiled and linked with your C files, with ESQL/C modules, and with 4GL libraries. This procedure also produces a customized Debugger for use with your application that calls the C functions or ESQL/C functions. The **cfgldb** command has the following structure.



Element	Description
cfgldb	A required keyword.
fgiusr.c	The name of the data file that you edited to declare your C and ESQL/C functions.
c filename	The name of a file containing one or more C functions that were declared in **fgiusr.c**. Three types of *c filenames* are recognized by **cfgldb**: ■ An object file (file extension **.o**) ■ A source file that contains C functions (file extension **.c**) ■ A source file that contains ESQL/C functions (file extension **.ec**)
-o newname	Specifies the name of the customized Debugger.
-V	The version number of the software.

Unless you specify the **-V** option, this creates a customized Debugger called *newname* that can execute C functions in the *c filename* files. These functions must be declared in the edited **fgiusr.c** file.

If you specify the **-V** option, the version numbers of your SQL and p-code compiler software are displayed, and the system prompt returns. No other output is produced, and any other options are ignored.

You can give the **fgiusr.c** file a different name. If you have renamed the **fgiusr.c** data file, you must substitute the new name in the **cfgldb** command line.

You can list the names of any number of C or ESQL/C files in any order, each separated by a blank space.

If you do not specify a name for the customized Debugger, the default name **a.out** is assigned.

Examples

The following command:

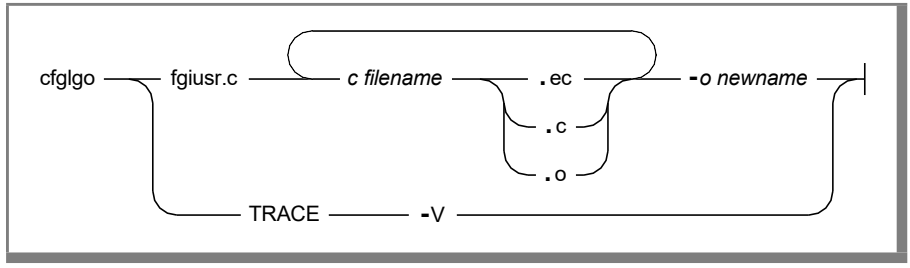
```
cfgldb fgiusr.c usr1.c usr2.ec usr3.o -o newdb
```

uses an edited **fgiusr.c** file to create a customized Debugger named **newdb** that can call C functions in the listed object file (**usr3.o**) as well as functions in a C source file (**usr1.c**) and ESQL/C (**usr2.ec**) functions. These functions must be declared in the edited **fgiusr.c** file.

Creating a Customized P-Code Runner

An edited **fgiusr.c** file can also be used to create a customized *p-code runner* for runtime execution of 4GL programs that call programmer-defined ESQL/C or C functions. The procedures to do this are outlined in this section.

Use the **cfglgo** facility to compile **fgiusr.c** and your ESQL/C or C modules to produce a customized p-code runner. The **cfglgo** command file has a syntax almost identical to that of **fgldb**.



Element	Description
cfglgo	A required keyword.
fgiusr.c	The name of the data file that you edited to declare your C and ESQL/C functions.
c filename	The name of a file containing one or more C functions that were declared in **fgiusr.c**. Three types of *c filenames* are recognized by **cfgldb**: ■ An object file (file extension **.o**) ■ A source file that contains C functions (file extension **.c**) ■ A source file that contains ESQL/C functions (file extension **.ec**)
-o newname	Specifies the name of the customized p-code runner.
-V	The version number of the software.

Unless you specify the **-V** option, this creates a customized p-code runner called *newname* that can execute C functions in the *c filename* files. These functions must be declared in the **fgiusr.c** file.

If you specify the **-V** option, the version numbers of your SQL and p-code compiler software are displayed, and the system prompt returns. No other output is produced, and any other options are ignored.

You can give a different name to the **fgiusr.c** file. If you have renamed the edited **fgiusr.c** data file, you must substitute the new name in the command line.

You can list the names of any number of C files or ESQL/C files in any order, each separated by a blank space.

If you do not specify a name for the customized p-code runner, the default name **a.out** is assigned.

Examples

The following command:

```
cfglgo fgiusr.c usr1.c usr2.ec usr3.o -o newgo
```

uses an edited **fgiusr.c** file to create a customized p-code runner named **newgo** that can call C functions from three files.

Using the fgiusr.c File with Several Programs

If you are developing several 4GL programs that call C functions, one way to use the **fgiusr.c** file is to create one version of the file for each application, with each file containing only those C functions that its application calls. You can then use the procedures described in the next section to create a different customized Debugger for each application that calls C functions.

If you want your users to execute the applications through a p-code runner, you can also create a customized p-code runner for each application that calls C functions. In this case, your users must know which customized p-code runner to use with each application program.

Alternatively, you can have one **fgiusr.c** file that declares and initializes all the C functions used in all your applications. You can then create a single modified Debugger and a single modified p-code runner that can execute all your C functions. Your users can then use the same customized p-code runner, regardless of the application. This is the preferred method.

Analyzing a 4GL Program That Calls C Functions

This section describes the steps that are required to analyze or run a 4GL program that calls C functions:

1. Be sure that 4GL and the Debugger are installed.
2. Use **fglpc** to compile your 4GL source code files into p-code.
3. For multi-module programs, concatenate the compiled **.4go** modules into a single **.4gi** file.
4. Edit the **fgiusr.c** file to declare your C functions.

5. Use **cfgldb** and the edited **fgiusr.c** file to compile your C functions and create a modified Debugger.
6. Invoke the modified Debugger to begin a debugging session.
7. After you are satisfied that the 4GL program is ready for your users, use **cfglgo** to create a customized p-code runner.

1. Environment Requirements

Make sure that you can access the required software. You cannot use the Debugger with a program that calls C functions unless you have 4GL and the Debugger correctly installed on your system, following the procedures that are described in the installation pamphlet. You must also have a C compiler.

The **PATH** environment variable tells the shell the correct search path for executable 4GL programs. The **INFORMIXDIR** and **DBPATH** environment variables are described in Appendix C, "Environment Variables," in the *INFORMIX-4GL Reference* manual. **DBSRC** is described in Appendix A of this manual. These environment variables must provide access from your current directory to all the necessary source, command, and data files.

2. Compile the 4GL Modules

If you have not already compiled your 4GL source modules, enter a command of the following form:

```
fglpc module1 [module2 ...]
```

Here *module1* and *module2* are the names of your 4GL source modules. This command produces a compiled **.4go** file for each **.4gl** source file. Chapter 8, "The Debugging Environment," describes the complete syntax of **fglpc**.

3. Concatenate the Compiled 4GL Modules

Unless your program includes only a single 4GL module, use the concatenation utility of your operating system to combine all the compiled modules in a single file. For example, a command of the form:

```
cat module1.4go module2.4go ... > filename.4gi
```

produces a single **.4gi** p-code file.

4. Edit the fgiusr.c File

The **fgiusr.c** file is a C source file in which you specify information that declares any programmer-defined C functions or ESQL/C functions that your 4GL program calls. Use a text editor to declare the name and the number of arguments of each C function.

The first 40 or so lines of the **fgiusr.c** file provide instructions. The unedited **fgiusr.c** file contains the following lines:

```
#include "fgicfunc.h"
cfunc_t  usrcfuncs[]  =
     {
     0,  0,  0
     };
```

Suppose, for example, that you want to call three C functions named **afunc1**, **bfunc2**, and **cfunc3**. Further suppose that the first requires one argument, the second requires none, and the third requires two arguments. To modify the **fgiusr.c** file so that it declares these functions, you must insert three new declaration statements:

```
#include "fgicfunc.h"
int afunc1();
int bfunc2();
int  cfunc3();
```

You must also insert lines to specify the number of arguments in each of the three functions before the line of zeroes. After you do this, the edited **fgiusr.c** file contains the following lines:

```
   #include "fgicfunc.h"
int afunc1();
int bfunc2();
int  cfunc3();
cfunc_t  usrcfuncs[]  =
     {
    "afunc1",  afunc1,  1,
    "bfunc2",  bfunc2,  0,
    "cfunc3",  cfunc3,  2,
     0,  0,  0
     };
```

When you have modified the file by providing this information, save it and return to the operating system prompt.

5. Create a Customized Debugger

Enter a command of the form:

```
cfgldb fgiusr.c   afunc1.c bfunc2.c cfunc3.c    -o dbfilename
```

where *afunc1.c*, *bfunc2.c*, and *cfunc3.c* are the names of source files that contain the C functions, and *dbfilename* is the name of the customized Debugger. The name of the output file can include the filename of the compiled 4GL program that you concatenated in step 3, if that helps you to remember that the customized Debugger is intended for use with that application.

Step 5 compiles the C functions and produces a customized Debugger. This modified version of the Debugger contains the C functions and ESQL/C functions from the modules that you listed in the command line. You also declared these functions in the **fgiusr.c** file.

6. Invoke the Debugger

You can now use the Debugger to examine the program. To begin a debugging session, enter:

```
dbfilename filename
```

where filename is the name of the output file from step 3, and *dbfilename* is the output file from step 5. The Debugger screen appears, with your 4GL program displayed in the Source window.

The syntax of a customized Debugger is the same as the syntax of **fgldb**. See "Invoking the Debugger" on page 8-11 for more information on the command-line options of a customized Debugger.

7. Create a Customized Runner for Your Program

Modify, recompile, and reanalyze your 4GL program to eliminate any errors. Then you can create a customized p-code runner, so your users can run the 4GL application that calls C functions.

To modify the p-code runner, you must perform the same first four steps that are required for using the Debugger with programs that call C functions. At step 5, however, create a customized p-code runner, rather than a Debugger, by entering a command of the following form:

```
cfglgo fgiusr.c   afile1.c bfile2.c ...    -o fnamefglgo
```

Here *fnamefglgo* is the name of the customized p-code runner, and *afile1.c* and *bfile2.c* are the names of the source files that contain the C functions. This list can also include object files (with extension **.o**) or ESQL/C files (with extension **.ec**).

The procedure compiles the C functions and produces a modified version of the p-code runner that can run your 4GL program that calls the specified C functions. You or your users can subsequently execute your compiled 4GL program by using the following command:

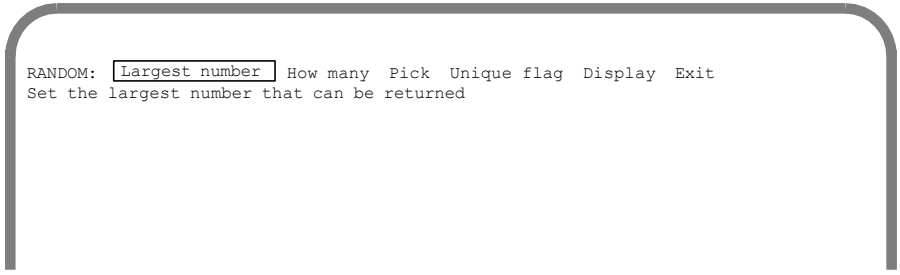
```
fnamefglgo filename
```

An Example of Calling a C Function

This section illustrates the procedure for using the Debugger with a 4GL program that calls C functions. The numbered subheadings correspond to the steps in the previous section. This section is designed as a tutorial. You use demonstration files to actually work through the procedure.

1. The Software Environment

This example creates a customized Debugger to examine a 4GL application that calls C functions. The application randomly selects an arbitrary number of numbers from a programmer-defined range of numbers. The program uses C functions to initialize the pseudo-random number generator, and to select and rescale pseudo-random numbers. The user selects options from the following menu, and the screen prompts for input after any choice except **Exit**.



Set the **PATH** and **INFORMIXDIR** environment variables to include the directory that contains your 4GL and Debugger command files, and your C compiler. Be sure that **DBSRC** specifies pathnames to the directories that hold the **r_globals.4gl** and **r_main.4gl** source files, and the C source files **fgiusr.c** and **getrand.c**, unless these files are in your current directory.

Important: *You can enter* `dbdemo` *to copy* ***r_globals.4gl***, ***r_main.4gl***, ***getrand.c****, and other demonstration files into your current directory. Use the appropriate operating system command to copy* ***fgiusr.c*** *from the* ***INFORMIXDIR/etc*** *directory to your current directory.*

A listing follows of the **r_main.4gl** module, which displays menu options from which the user can select random numbers:

```
 1 GLOBALS
 2     "r_globals.4gl"
 3
 4 MAIN
 5
 6    LET largest = 49
 7    LET howmany = 6
 8    LET unique_flag = TRUE
 9    CALL initrand()
10
11    MENU "RANDOM"
12      COMMAND "Largest_number"
13        "Set the largest number that can be returned"
14          CALL setlargest()
15      COMMAND "How_many" "Set how many numbers to pick"
16          CALL sethowmany()
17      COMMAND "Pick" "Generate a set of random numbers"
18          CALL getpick()
19      COMMAND "Unique_flag" "Set or Unset the unique flag"
20          CALL setunique()
21      COMMAND "Display" "Display set variables"
22          CALL dispvars()
23      COMMAND "Exit" "Exit program"
24          CLEAR SCREEN
25          EXIT PROGRAM
26    END MENU
27 END MAIN
28
29
30 FUNCTION setlargest()
31
32    CALL clear_menu()
33    WHILE TRUE
34      PROMPT "Enter Largest Number: " FOR largest
35      IF largest > 0 THEN
36          RETURN
37      END IF
38
39      DISPLAY "Number must be greater than zero"
40    END WHILE
41
42 END FUNCTION
```

```
 43
 44
 45 FUNCTION sethowmany()
 46
 47    CALL clear_menu()
 48    WHILE TRUE
 49      PROMPT "Enter How Many Numbers to Pick: " FOR howmany
 50      IF howmany > 0 AND howmany < 101 THEN
 51          RETURN
 52      END IF
 53
 54      DISPLAY "Number must be between 1 and 100"
 55    END WHILE
 56
 57 END FUNCTION
 58 FUNCTION getpick()
 59
 60    DEFINE i INTEGER,
 61          rnum INTEGER,
 62          x CHAR(1)
 63
 64    CALL clear_menu()
 65
 66    IF unique_flag = TRUE AND largest < howmany THEN
 67      DISPLAY "ERROR - Cannot find ", howmany using "<<<",
 68        " unique numbers less than or equal to ",
 69        largest using "<<<<<<<<<<<<<<"
 70      SLEEP 3
 71      RETURN
 72    END IF
 73
 74    FOR i =1  TO howmany
 75      LET rnum = getrand(largest)
 76      IF unique_flag = TRUE THEN
 77         WHILE alreadypicked(rnum, i)
 78            LET rnum = getrand(largest)
 79         END WHILE
 80      END IF
 81      LET picked[i] = rnum
 82      DISPLAY picked[i]
 83    END FOR
 84
 85    PROMPT "Press any key to continue" FOR CHAR x
 86
 87 END FUNCTION
 88
 89
 90 FUNCTION alreadypicked(rnumber, cnt)
 91   DEFINE rnumber INTEGER,
 92        cnt INTEGER,
 93        i INTEGER
 94
 95   FOR i =1  to cnt
 96        IF picked[i] = rnumber THEN
 97           RETURN TRUE
 98        END IF
 99   END FOR
100
101   RETURN FALSE
102
103 END FUNCTION
```

```
104
105
106 FUNCTION setunique()
107
108   CALL clear_menu()
109   MENU "UNIQUE"
110      COMMAND "Yes" "Numbers must be unique"
111         LET unique_flag = TRUE
112         CALL mess("Unique flag is set", 23)
113      COMMAND "No" "Numbers do not have to be unique"
114         LET unique_flag = FALSE
115         CALL mess("Unique flag is no longer set", 23)
116      COMMAND "Exit" "Return to random menu"
117         Exit MENU
118   END MENU
119 END FUNCTION
120
121
122 FUNCTION mess(str, mrow)
123   DEFINE str CHAR(80),
124         mrow SMALLINT
125
126    DISPLAY " ", str CLIPPED AT mrow,1
127    SLEEP 3
128    DISPLAY "" AT mrow,1
129 END FUNCTION
130
131
132 FUNCTION dispvars()
133
134    DEFINE x CHAR(1)
135
136    CALL clear_menu()
137    display "LARGEST NUMBER:    ", largest
138    display "HOW MANY NUMBERS: ", howmany
139    IF unique_flag = TRUE THEN
140       display "Unique flag is set"
141    ELSE
142       display "Unique flag is not set"
143    END IF
144
145    PROMPT "Press any key to continue" FOR CHAR x
146
147 END FUNCTION
148
149
150 FUNCTION clear_menu()
151
152    DISPLAY "" AT 1,1
153    DISPLAY "" AT 2,1
154 END FUNCTION
```

The r_globals.4gl Module

A listing follows of the **r_globals.4gl** module, which is called by the **r_main.4gl** module:

```
1 GLOBALS
2     DEFINE  largest  INTEGER,
3        howmany  INTEGER,
4        picked  ARRAY[100]  of  INTEGER,
5        unique_flag  INTEGER
6 END  GLOBALS
```

The getrand.c Module

The next listing shows the **getrand.c** module. The first function, **initrand**, uses the time of day to initialize the random number generator. The second C function, **getrand**, uses the C library random number generator to select a set of numbers and rescales the returned value to conform to specifications supplied by the user. The first function requires no argument, and the second requires one.

```
 1 /*
 2  * Use the time of day to initialize
 3  * the random number generator.
 4 */
 5 initrand(numargs)
 6 int numargs; /* number of 4GL parameters passed */
 7{
 8      int timeofday;
 9
10      /*
11       * Get the current time in seconds and use that
12       * as the seed to the random number generator.
13       */
14      timeofday = time(0);
15      srandom(timeofday);
16      return(0);
17 }
18
19 /*
20  * Get the next random number
21  */
22 getrand(numargs)
23 int numargs;/* number of 4GL parameters passed */
24 {
25     int rnumber; /* random number generated */
26     int maximum; /* largest number that
27                     * can be generated */
28
29     /*
30      * Make sure the maximum number which can be
31      * generated was passed.
32      * Pop the argument off the stack into maximum.
33      */
```

```
34     if (numargs != 1)
35           exit(-1);
36     popint(&maximum);
37
38     /*
39      * Get the random number, then use modulo arithmetic
40      * to make sure the number isn't larger than maximum.
41      * NOTE:  random returns a number
42      * between 0 and (2 to the 31) - 1
43      */
44    rnumber = random();
45    rnumber = (rnumber % maximum) + 1;
46
47    retint(rnumber); /* put the random number on the stack */
48    return(1);
49 }
```

2. Compiling the r_globals.4gl and r_main.4gl Modules

To create compiled versions of the 4GL source modules, enter the following command at the operating system prompt:

```
fglpc  r_globals  r_main
```

This produces two output files, **r_globals.4go** and **r_main.4go**.

3. Concatenating r_globals.4go and r_main.4go

To combine the compiled modules, enter:

```
cat  r_globals.4go  r_main.4go  >  newfile.4gi
```

The filename of the output file (**newfile**) must appear later in the command lines of step 6 and step 7.

4. Editing the fgiusr.c File

The unedited copy of the **fgiusr.c** file was listed earlier in this appendix. You must insert into this file the names and the number of arguments of each C function, namely **initrand** and **getrand**.

Declaring two functions requires that you use an editor to insert four new lines near the end of the **fgiusr.c** file. Invoke an editor, and modify the **fgiusr.c** file to conform to the following listing:

```
/*********************************************************
*                                                        *
*            INFORMIX  SOFTWARE,  INC.                   *
*                                                        *
* Title:  fgiusr.c                                       *
* Sccsid:  @(#)fgiusr.c     4.2    6/26/99   10:48:37     *
* Description:                                           *
*         definition  of  user  C  functions             *
*                                                        *
*********************************************************
*/

/*****************************************************
* This  table  is  for  user-defined  C  functions.
*
* Each  initializer  has  the  form:
*
*   "name",  name,  nargs
*
* Variable  #  of  arguments:
*
*   set  nargs  to  -(maximum  #  args)
*
* Be  sure  to  declare  name  before  the  table  and  to  leave  the
* line  of  0's  at  the  end  of  the  table.
*
* Example:
*
*
You  want  to  call  your  C  function  named "mycfunc" and it expe
cts
*   2  arguments.   You  must  declare  it:
*
*        int  mycfunc();
*
*   and  then  insert  an  initializer  for  it  in  the  table:
*
*        "mycfunc",  mycfunc,  2
*****************************************************
*/
*/

#include "fgicfunc.h"
int initrand();
int  getrand();
```

```
cfunc_t  usrcfuncs[]  =
     {
     "initrand",  initrand,  0,
     "getrand",  getrand,  1,
     0,  0,  0
     };
```

5. Compiling initrand.c and getrand.c

This step compiles the files that contain the C functions and creates a modified Debugger. Do this at the system prompt by entering the command:

```
cfgldb  fgiusr.c  getrand.c  -o  newdb
```

This creates a customized Debugger called **newdb** that can be used in a debugging session. This customized Debugger can execute the compiled and concatenated 4GL program file **newfile.4gi** that contains the output from step 3.

6. Debugging a Program That Calls C Functions

To invoke the customized Debugger that you created in step 5, enter the following command at the operating system prompt:

```
newdb  newfile
```

The screen displays the Source and Command windows, with statements from the source file **r_main.4gl** displayed in the Source window. You can enter a RUN or CALL command to see output from this program in the Application screen.

7. Creating a Customized P-Code Runner

If you are satisfied that this application is ready for your users, you can create a customized p-code runner so that they can execute the program. To do so, enter the following command at the operating system prompt:

```
cfglgo  fgiusr.c  getrand.c  -o  newfglgo
```

As in step 5, the name that you assign to the output file is arbitrary. Now you can execute the 4GL program that calls C functions by invoking the customized p-code runner as follows:

```
newfglgo  newfile
```

How C Functions Affect Debugger Commands

This section describes some differences between debugging an ordinary 4GL program and debugging one that calls C functions. These differences have no effect on how the Debugger interacts with 4GL statements, but they do prevent you from displaying C functions. The most important differences are listed here:

- You can use CALL, CONTINUE, STEP, and RUN commands to execute C functions, but you cannot display C source code in the Source window. If you enter a VIEW command to display a C function, an error message appears. To look at C source code, you must use the Escape feature to enter an appropriate operating system command.
- Because C functions contain no executable 4GL statements, you cannot use the STEP INTO command to execute individual statements within a C function. The 4GL statement that calls the C function, together with the function itself, are treated as a single statement by the STEP command.
- You cannot set a breakpoint or tracepoint inside a C function. You can specify the name of a C function in a TRACE command but not in a BREAK command. You can use BREAK or TRACE, however, at the line number of a 4GL statement that calls a C function.
- The DUMP, PRINT, and VARIABLE commands do not describe variables that exist only in C functions. The FUNCTIONS and WHERE commands do not return the name of any C function. (If you trace the name of a C function, however, a WHERE command in the tracepoint specification shows the name of the C function, any values passed with it, and the functions currently active.)

In general, the Debugger allows you to display source code and trace the execution of a 4GL program up to a C function call and after a C function call, but not inside a C function. By following the procedures in this appendix, however, you can use Debugger commands to execute C functions and ESQL/C functions that are called from 4GL programs.

Appendix

C

Sample Programs

This appendix describes the sample programs used in this manual to demonstrate the INFORMIX-4GL Interactive Debugger. The **customer** program demonstrates basic debugging techniques (Appendix 3, "Tracing Logic of the customer Program," and Chapter 4, "Analyzing a Logical Error in the customer Program"), while the **cust_order** program is the basis for advanced debugging sessions (Appendix 6, "Tracing Logic of the cust_order Program," and Chapter 7, "Analyzing Runtime Errors in the cust_order Program").

Both programs, as supplied with the Debugger, contain intentional errors. This appendix documents the *debugged* versions of the programs.

The customer Program

This section describes the **customer** program, a single-module program that uses the **customer** form to add, retrieve, modify, and delete customer rows from the database.

The **customer** program consists of the following program blocks and functions:

```
GLOBALS
MAIN
show_menu
enter_row
query_data
change_data
delete_row
```

As explained in Chapter 4, "Analyzing a Logical Error in the customer Program," the original version of the **customer** program contains errors. This appendix discusses the debugged version of the program.

In the program listing, some functions are annotated to help you locate individual statements. The numbers in the listing correspond to numbered items in the text that describes the function.

Program Listing

This section lists the debugged version of the **customer** program:

```
DATABASE stores7
GLOBALS
    DEFINE
       p_customer RECORD LIKE customer.*,
       chosen SMALLINT
END GLOBALS
MAIN
    DEFER  INTERRUPT
    OPEN FORM cust_form FROM "customer"
    DISPLAY FORM cust_form
    LET chosen = FALSE
    OPTIONS MESSAGE LINE 22,
        PROMPT  LINE  21,
        HELP FILE "custhelp.ex",
        HELP KEY CONTROL-I
    CALL show_menu()
    MESSAGE "End program."
    SLEEP 3
```

```
    CLEAR SCREEN
END MAIN
FUNCTION  show_menu()
    DEFINE answer  CHAR(1)
    MESSAGE "Type the first letter of the option ",
       "you want to select or CONTROL I for Help."
    MENU "CUSTOMER"
    COMMAND "Add" "Add a new customer." HELP 1
       LET answer = "y"
          WHILE answer = "y"
             CALL enter_row()
             PROMPT  "Do  you  want  to  ",
                "enter another row (y/n) ?  "
                FOR CHAR answer
          END WHILE
       CLEAR FORM
    COMMAND "Query" "Search for a customer." HELP 2
       CALL query_data()
       IF  chosen  THEN
          NEXT OPTION "Modify"
       END IF
    COMMAND "Modify" "Modify a customer." HELP 3
       IF chosen THEN
           CALL  change_data()

       ELSE
          MESSAGE "No customer has been chosen. ",
             "Use the Query option to select ",
             "a customer."
          NEXT OPTION "Query"
       END IF
    COMMAND "Delete" "Delete a customer." HELP 4
       IF chosen THEN
          PROMPT "Are you sure you want to ",
             "delete this customer (y/n)?  "
             FOR CHAR answer
             IF answer = "y" THEN
                CALL delete_row()
                LET chosen = FALSE
              END IF
       ELSE
          MESSAGE "No customer has been chosen. ",
             "Use the Query option to select ",
             "a customer."
          NEXT OPTION "Query"
       END IF
    COMMAND "Exit" "Leave the CUSTOMER menu." HELP 5
    EXIT MENU
    END MENU
END FUNCTION
FUNCTION enter_row()
    LET int_flag = 0
    MESSAGE ""
    CLEAR  FORM
```

```
    INPUT p_customer.fname THRU p_customer.phone
       FROM sc_cust.*
    IF int_flag THEN
       LET int_flag = FALSE
       ERROR "Customer entry aborted."
       RETURN
    END IF
    LET p_customer.customer_num =    0
    INSERT INTO customer VALUES (p_customer.*)
    LET p_customer.customer_num = SQLCA.SQLERRD[2]
    DISPLAY p_customer.customer_num TO customer_num
    MESSAGE "Row added."
    SLEEP 3
    MESSAGE ""
END FUNCTION
FUNCTION query_data()
    DEFINE
       where_clause CHAR(200),
       sql_stmt CHAR(250),
       answer CHAR(1),
       exist SMALLINT
    LET int_flag = 0
    MESSAGE ""
    CLEAR FORM
    MESSAGE "Enter search criteria and press ESC."
    SLEEP 3
    MESSAGE  ""
    CONSTRUCT where_clause on customer.* FROM customer_num,
      fname, lname, company, address1, address2, city, state,
      zipcode, phone
    IF int_flag THEN
       LET int_flag = FALSE
       ERROR "Customer query aborted"
       RETURN
    END IF
    LET sql_stmt = "SELECT * FROM customer where ",
       where_clause clipped
    PREPARE ex_sel FROM sql_stmt
    DECLARE q_curs CURSOR FOR ex_sel
    LET exist = FALSE
    LET chosen = FALSE
    FOREACH q_curs INTO p_customer.*
       LET exist = TRUE
       DISPLAY BY NAME p_customer.*
       PROMPT "Enter 'y' to select this customer ",
          "or RETURN to view next customer:     "
          FOR CHAR answer
          IF answer = "y" THEN
             LET chosen = TRUE
             EXIT FOREACH
          END IF
    END FOREACH
    IF exist = FALSE THEN
       MESSAGE "No customer rows found."
```

```
        SLEEP 3
        MESSAGE ""
    ELSE
        IF chosen = FALSE THEN
            MESSAGE "There are no more customer rows."
            SLEEP 3
            MESSAGE ""
            CLEAR FORM
        END IF
    END IF
END FUNCTION
FUNCTION change_data()
    LET int_flag = 0
    INPUT p_customer.fname THRU p_customer.phone
        WITHOUT DEFAULTS FROM sc_cust.*
    IF int_flag THEN
        LET int_flag = FALSE
        ERROR "Customer update aborted."
        RETURN
    END IF
    UPDATE customer
        SET customer.* = p_customer.*
        WHERE customer_num = p_customer.customer_num
    MESSAGE "Row updated."
    SLEEP 3
    MESSAGE ""
END FUNCTION
FUNCTION delete_row()
    LET int_flag = 0
    DELETE FROM customer WHERE customer_num =
        p_customer.customer_num
    CLEAR FORM
    MESSAGE "Row deleted."
    SLEEP 3
    MESSAGE ""
END FUNCTION
```

The GLOBALS Statement

The GLOBALS section of the program defines two global variables:

- **p_customer**, a record that contains variables corresponding to the columns in the **customer** table
- **chosen**, a flag indicating whether the user has selected a customer

The MAIN Statement

The MAIN section of the program performs the following operations:

1. Defers interrupt signals so program execution will continue if the user presses the **Interrupt** key (typically CTRL-C)
2. Opens and displays the **customer** form
3. Sets **chosen** to FALSE to indicate that no customer has yet been selected
4. Establishes line 22 as the message line and line 21 as the prompt line, specifies the pathname of the help file, and designates CTRL-I as the help key
5. Calls the **show_menu** function, which contains the MENU statement for the **CUSTOMER** menu
6. Displays a message to indicate that the program is over and clears the screen

The show_menu Function

The **show_menu** function sets up the **CUSTOMER** menu and calls the functions that carry out actions described by the menu options.

The **show_menu** function performs the following operations:

1. Displays a message to tell the user how to choose an option or display a help message
2. Displays the menu name, menu options, and the help line for the highlighted option, as specified in the MENU statement

3. Executes the Add COMMAND clause when the user chooses **Add** from the menu

   If the user presses CTRL-I while the **Add** option is highlighted, the function displays help message 1.

   The Add COMMAND clause performs the following operations:

   a. Assigns the value `y` to the variable **answer**

   b. Executes the statements in the WHILE loop because the initial value of **answer** is `y`

   c. Calls the **enter_row** function that allows the user to enter a row on the **customer** form and then inserts the row into the **customer** table

   d. Prompts the user to supply another value for **answer**

   e. If the user enters a value other than `y`, leaves the WHILE loop and redisplays the menu so that the user can select another option

   f. If the user enters `y`, executes the statements in the WHILE loop again

4. Executes the Query COMMAND clause when the user chooses the **Query** option

   If the user presses CTRL-I while the **Query** option is highlighted, the program displays help message 2.

   The Query COMMAND clause performs the following operations:

   a. Calls the **query_data** function so the user can query for a customer

      The value of the global variable **chosen** is changed to TRUE if the user selects a customer from among the query results.

   b. Highlights the **Modify** option if a customer has been chosen

5. Executes the Modify COMMAND clause when the user chooses the **Modify** option

   If the user presses CTRL-I while the **Modify** option is highlighted, the program displays help message 3.

   The Modify COMMAND clause performs the following operations:

   a. If **chosen** is TRUE (indicating that the user has selected a customer), the program calls the **change_data** function, which allows the user to update the current row.

   b. Otherwise, the program displays a message indicating that no customer row has been chosen and highlights the **Query** option.

6. Executes the Delete COMMAND clause when the user selects **Delete** from the menu

   If the user presses CTRL-I while the **Delete** option is highlighted, the program displays help message 4.

   The Delete COMMAND clause performs the following operations:

   a. If **chosen** is TRUE (indicating that the user has selected a customer), the program asks the user if the current row should be deleted. If the user enters `y`, the program calls the **delete_row** function and then resets **chosen** to FALSE.

   b. Otherwise, the program displays a message indicating that no customer row has been chosen and highlights the **Query** option.

7. Executes the Exit COMMAND clause when the user selects the **Exit** option

   If the user presses CTRL-I while the highlight is on the **Exit** option, the program displays help message 5. The Exit COMMAND clause terminates the MENU statement.

The enter_row Function

The **enter_row** function performs the following operations:

1. Sets the **int_flag** global variable to `0` (FALSE) to ensure the validity of a later interrupt test
2. Clears the message line and form

3. Assigns values to all variables in the **p_customer** record except **p_customer.customer_num** from the data entered by the user on the screen form
4. Tests the value of the global variable **int_flag**

   With interrupts deferred, the flag is set to nonzero when the user presses the Interrupt key, and the program continues. If the interrupt test is true, the program responds as follows:

   a. Resets **int_flag** to FALSE to ensure that later interrupt tests are valid
   b. Displays the message `Customer query aborted` on the error message line
   c. Exits the function
5. Assigns the value `0` to **p_customer.customer_num** as a placeholder for the serial value that 4GL will insert automatically
6. Inserts the data in **p_customer** into the **customer** table
7. Assigns the value in SQLCA.SQLERRD[2] to **p_customer.customer_num**

   SQLCA.SQLERRD[2] stores the serial value of the row that the program just inserted into the **customer** table.
8. Displays the serial number for the row just added to the **customer** table, along with the message `Row added`

The query_data Function

The **query_data** function selects customer rows based on a last name that the user supplies. The function performs the following operations:

1. Sets the **int_flag** global variable to `0` (FALSE) to ensure the validity of a later interrupt test
2. Clears the message line and screen form
3. Displays and then clears the message `Enter search criteria and press ESC`

4. Uses a CONSTRUCT statement to allow the user to perform a query by example on the **customer** table:

```
CONSTRUCT where_clause on customer.* FROM customer_num,
  fname, lname, company, address1, address2, city, state,
  zipcode, phone
```

   The user can now enter selection criteria in selected fields of the **customer** screen record. When the user presses ESC, the selection criteria are stored in the local variable **where_clause**.

5. Uses an IF statement to test the value of the global variable **int_flag**

   With interrupts deferred, this flag is set to nonzero when the user presses the Interrupt key, and the program continues. If the interrupt test is true, the program does the following:

   a. Resets **int_flag** to FALSE to ensure that later interrupt tests are valid

   b. Displays the message `Customer query aborted` on the error message line

   c. Exits the function

6. Concatenates **where_clause** to a character string that contains the rest of the SELECT statement and assigns the result to the **sql_stmt** local variable

7. Prepares an executable statement called **ex_sel** from the SELECT statement stored in **sql_stmt**

8. Declares a cursor for a SELECT statement that retrieves all customers that meet the user's selection criteria

9. Initializes the local variable **exist** to FALSE, indicating that no rows have yet been found

10. Sets the global variable **chosen** to FALSE to indicate that the user has not yet selected a customer

11. Uses a FOREACH loop to display customer information on the screen form

    The FOREACH loop performs the following operations:

    a. Retrieves a customer row from the **customer** table and stores it in the record **p_customer**

    b. Assigns the value TRUE to the local variable **exist** to indicate that at least one customer row has been retrieved

    c. Displays the values in **p_customer** on the screen form

    d. Asks whether the user wants to select the current customer or view the next customer

       If the user enters `y` in response to the prompt, the program sets **chosen** to TRUE and exits the FOREACH loop.

    e. Repeats steps a through d until all customer rows are processed, or until the user enters `y` to choose a customer and exit the FOREACH statement

    a. Displays a message if no customer rows were found (**exist** is FALSE)

12. Displays a message and clears the form if the user wanted to see another customer (**chosen** is FALSE), but the program could not find one

The change_data Function

The **change_data** function enables the user to change a customer row. The program calls the **change_data** function only if the current value of the **chosen** variable is TRUE, indicating that the user has selected a customer row using the **query_data** function.

The **change_data** function performs the following operations:

1. Sets the **int_flag** global variable to `0` (FALSE) to ensure the validity of a later interrupt test
2. Uses the INPUT statement WITHOUT DEFAULTS to assign values to all variables in **p_customer** record except **p_customer.customer_num** from data on the screen form

3. Uses an IF statement to test the value of the global variable **int_flag**

   With interrupts deferred, this flag is set to nonzero when the user presses the Interrupt key, and the program continues. If the interrupt test is true, the program responds as follows:

   a. Resets **int_flag** to FALSE to ensure that later interrupt tests are valid

   b. Displays the message `Customer update aborted` on the error message line

4. Updates the row in the **customer** table where the value in the **customer_num** column equals the value currently stored in **p_customer.customer_num**
5. Displays a message indicating that the row has been updated

The delete_row Function

The **delete_row** function deletes the currently displayed row from the **customer** table. The program calls the **delete_row** function only if the current value of the **chosen** variable is TRUE, indicating that the user has selected a customer row using the **query_data** function.

The **delete_row** function performs the following operations:

1. Sets the **int_flag** global variable to `0` (FALSE) to ensure the validity of a later interrupt test
2. Deletes the row in the **customer** table where the value in the **customer_num** column equals the value currently stored in **p_customer.customer_num**
3. Clears the form
4. Informs the user that the row has been deleted

The cust_order Program

This section describes the **cust_order** program, a multi-module program that uses the **orderform** form to add customer orders to the **stores7** database.

The **cust_order** program consists of the following modules and functions.

Module 1	Module 2	Module 3
GLOBALS	GLOBALS	GLOBALS
	MAIN	add_order
	mess	insert_order
	clear_menu	order_total
	fetch_stock	item_total
	query_customer	renum_items
		insert_items
		get_stock
		get_item
		find_order

As explained in Chapter 6, "Tracing Logic of the cust_order Program," the original version of the **cust_order** program contains fatal errors. This appendix discusses the debugged version of the program.

Attributes, as in the following DISPLAY FORM statement, are not discussed unless they affect the operation of the program:

```
DISPLAY FORM order_form
   ATTRIBUTE(MAGENTA)
```

For information about attributes, see the *INFORMIX-4GL Reference* manual.

In the program listing, some functions are annotated to help you locate individual statements. The numbers in the listing correspond to numbered items in the text that describes the function.

Program Listing

This section lists the debugged versions of all three modules in the **cust_order** program.

Module 1: globals.4gl

The globals.4gl module contains the following code:

```
DATABASE stores7

GLOBALS
    DEFINE
        p_customer RECORD LIKE customer.*,
        p_orders RECORD
            order_num LIKE orders.order_num,
            order_date  LIKE  orders.order_date,
            po_num   LIKE   orders.po_num,
            ship_instruct  LIKE  orders.ship_instruct
         END  RECORD,
         p_items  ARRAY[10]  OF  RECORD
            item_num LIKE items.item_num,
            stock_num LIKE items.stock_num,
            manu_code LIKE items.manu_code,
            description LIKE stock.description,
            quantity LIKE items.quantity,
            unit_price LIKE stock.unit_price,
            total_price  LIKE  items.total_price
          END  RECORD,
          p_stock  ARRAY[15]  OF  RECORD
              stock_num LIKE stock.stock_num,
              manu_code LIKE manufact.manu_code,
              manu_name LIKE manufact.manu_name,
              description LIKE stock.description,
              unit_price LIKE stock.unit_price,
              unit_descr  LIKE  stock.unit_descr
          END  RECORD,
              stock_cnt INTEGER
END GLOBALS
```

Module 2: main.4gl

The main.4gl module contains the following code:

```
GLOBALS
   "globals.4gl"

MAIN

   DEFER INTERRUPT

   OPEN FORM order_form FROM "orderform"
   DISPLAY FORM order_form
      ATTRIBUTE(MAGENTA)
   MENU "ORDERS"
      COMMAND "Add-order"
     "Enter new order to database"
         CALL add_order()
      COMMAND "Find-order" "Look up and display orders"
         CALL find_order()
      COMMAND "Exit" "Exit program and return to operating system"
         CLEAR SCREEN
         EXIT PROGRAM
   END MENU

END MAIN

FUNCTION mess(str, mrow)
   DEFINE str CHAR(80),
       mrow SMALLINT

   DISPLAY " ", str CLIPPED AT mrow,1
   SLEEP 3
   DISPLAY "" AT mrow,1
END FUNCTION

FUNCTION clear_menu()

   DISPLAY "" AT 1,1
   DISPLAY "" AT 2,1
END FUNCTION

FUNCTION fetch_stock()

   DECLARE stock_list CURSOR FOR
   SELECT stock_num, manufact.manu_code,
      manu_name, description, unit_price, unit_descr
      FROM stock, manufact
      WHERE stock.manu_code = manufact.manu_code
      ORDER BY stock_num
   LET stock_cnt = 1
   FOREACH stock_list INTO p_stock[stock_cnt].*
      LET stock_cnt = stock_cnt + 1
      IF stock_cnt > 15 THEN
         EXIT FOREACH
      END IF
    END FOREACH
    LET stock_cnt = stock_cnt - 1
END FUNCTION
```

```
FUNCTION query_customer(mrow)
   DEFINE where_part CHAR(200),
      query_text CHAR(250),
      answer CHAR(1),
      mrow, chosen, exist SMALLINT

   CLEAR FORM
   CALL clear_menu()
   MESSAGE "Enter criteria for selection"
   CONSTRUCT where_part ON customer.* FROM customer.*
   MESSAGE ""
   IF int_flag THEN
      LET int_flag = FALSE
      CLEAR FORM
      ERROR "Customer query aborted" ATTRIBUTE(RED, REVERSE)
      LET p_customer.customer_num = NULL
      RETURN (p_customer.customer_num)
   END IF
   LET query_text = "select * from customer where ",
      where_part CLIPPED,
      "order by lname"
   PREPARE statement_1 FROM query_text
   DECLARE customer_set SCROLL CURSOR FOR statement_1

   OPEN customer_set
   FETCH FIRST customer_set INTO p_customer.*
   IF status = NOTFOUND THEN
      LET exist = FALSE
   ELSE
      LET exist = TRUE
      DISPLAY BY NAME p_customer.* ATTRIBUTE(MAGENTA)
      MENU "BROWSE"
         COMMAND "Next" "View the next customer in the list"
            FETCH NEXT customer_set INTO p_customer.*
               IF status = NOTFOUND THEN
                  ERROR "No more customers in this direction"
                     ATTRIBUTE(RED, REVERSE)
                  FETCH LAST customer_set INTO p_customer.*
               END IF
            DISPLAY BY NAME p_customer.* ATTRIBUTE(MAGENTA)
         COMMAND "Previous" "View the previous customer in the list"
            FETCH PREVIOUS customer_set INTO p_customer.*
               IF status = NOTFOUND THEN
                  ERROR "No more customers in this direction"
                     ATTRIBUTE(RED, REVERSE)
                  FETCH FIRST customer_set INTO p_customer.*
               END IF
            DISPLAY BY NAME p_customer.* ATTRIBUTE(MAGENTA)
         COMMAND "First" "View the first customer in the list"
            FETCH FIRST customer_set INTO p_customer.*
            DISPLAY BY NAME p_customer.* ATTRIBUTE(MAGENTA)
         COMMAND "Last" "View the last customer in the list"
            FETCH LAST customer_set INTO p_customer.*
            DISPLAY BY NAME p_customer.* ATTRIBUTE(MAGENTA)
         COMMAND "Select" "Exit BROWSE selecting the current customer"
            LET chosen = TRUE
            EXIT MENU
         COMMAND "Quit" "Quit BROWSE without selecting a customer"
```

```
            LET chosen = FALSE
            EXIT MENU
      END MENU
   END IF
   CLOSE customer_set

   CALL clear_menu()
   IF NOT exist THEN
      CLEAR FORM
      CALL mess("No customer satisfies query", mrow)
      LET p_customer.customer_num = NULL
      RETURN (FALSE)
   END IF
   IF NOT chosen THEN
      CLEAR FORM
      CALL mess("No selection made", mrow)
      LET p_customer.customer_num = NULL
      RETURN (FALSE)
   END        IF
   RETURN (TRUE)
END FUNCTION
```

Module 3: order.4gl

The order.4gl module contains the following code:

```
GLOBALS
    "globals.4gl"

   DEFINE query_stat INTEGER

FUNCTION add_order()
   DEFINE pa_curr, s_curr INTEGER

   LET query_stat = query_customer(2)
   IF query_stat IS NULL OR query_stat = 0 THEN
      RETURN
   END IF
   DISPLAY by name p_customer.* ATTRIBUTE(CYAN)

   MESSAGE "Enter the order date, PO number and shipping instructions."
   INPUT BY NAME p_orders.order_date, p_orders.po_num,
      p_orders.ship_instruct
   IF int_flag THEN
      LET int_flag = FALSE
      CLEAR FORM
      ERROR "Order input aborted" ATTRIBUTE (RED, REVERSE)
      RETURN
   END IF
   INPUT ARRAY p_items FROM s_items.*
      BEFORE FIELD stock_num
      MESSAGE "Press ESC to write order"
      DISPLAY "Enter a stock number or press CTRL-B to scan stock list"
         AT 1,1
      BEFORE FIELD manu_code
         MESSAGE "Press ESC to write order"
         DISPLAY "" AT 1, 1
         DISPLAY "Enter a manufacturer code or press CTRL-B to scan ",
            "stock list" at 1, 1
```

```
        BEFORE FIELD quantity
           MESSAGE "Press ESC to write order"
           DISPLAY "" AT 1,1
           DISPLAY "Enter the item quantity" AT 1, 1
           ON KEY (CONTROL-B)
              IF INFIELD(stock_num) OR INFIELD(manu_code) THEN
                 LET pa_curr = arr_curr()
                 LET s_curr = scr_line()
                 CALL fetch_stock()
                 CALL get_stock() RETURNING
                    p_items[pa_curr].stock_num, p_items[pa_curr].manu_code,
                    p_items[pa_curr].description,  p_items[pa_curr].unit_price
                 DISPLAY p_items[pa_curr].stock_num
                    TO s_items[s_curr].stock_num
                 DISPLAY p_items[pa_curr].manu_code
                    TO s_items[s_curr].manu_code
                 DISPLAY p_items[pa_curr].description
                    TO s_items[s_curr].description
                 DISPLAY p_items[pa_curr].unit_price
                    TO s_items[s_curr].unit_price
                 NEXT FIELD quantity
              END IF
        AFTER FIELD stock_num, manu_code
           LET pa_curr = arr_curr()
           IF p_items[pa_curr].stock_num IS NOT NULL
              AND p_items[pa_curr].manu_code IS NOT NULL
           THEN
              CALL get_item()
           END IF
        AFTER FIELD quantity
           MESSAGE ""
           LET pa_curr = arr_curr()
           IF p_items[pa_curr].unit_price IS NOT NULL
              AND p_items[pa_curr].quantity IS NOT NULL
           THEN
              CALL item_total()
           ELSE
              ERROR "A valid stock code, manufacturer, and ",
                 "quantity must all be entered" ATTRIBUTE (RED, REVERSE)
              NEXT FIELD stock_num
           END IF
        AFTER INSERT, DELETE
           CALL renum_items()
           CALL order_total()
        AFTER ROW
           CALL order_total()
     END INPUT

     IF int_flag THEN
        LET int_flag = FALSE
        CLEAR FORM
        ERROR "Order input aborted" ATTRIBUTE (RED, REVERSE)
        RETURN
     END IF

     CALL insert_order()
  END FUNCTION

  FUNCTION insert_order()
     WHENEVER ERROR CONTINUE
     BEGIN WORK
```

```
    INSERT INTO orders (order_num, order_date, customer_num,
       ship_instruct, po_num)
       VALUES (0, p_orders.order_date, p_customer.customer_num,
          p_orders.ship_instruct, p_orders.po_num)
       IF status < 0 THEN
          ROLLBACK WORK
          ERROR "Unable to complete update of orders table"
          ATTRIBUTE(RED, REVERSE, BLINK)
          RETURN
       END IF
    LET p_orders.order_num = SQLCA.SQLERRD[2]
    DISPLAY BY NAME p_orders.order_num
       IF NOT insert_items() THEN
          ROLLBACK WORK
          ERROR "Unable to insert items" ATTRIBUTE(RED, REVERSE, BLINK)
          RETURN
       END IF
    COMMIT WORK
    WHENEVER ERROR STOP
    CALL mess("Order added", 23)
    CLEAR FORM
END FUNCTION

FUNCTION order_total()
    DEFINE order_total MONEY(8),
       i INTEGER
    LET order_total = 0.00
    FOR i = 1 TO arr_count()
       IF p_items[i].total_price IS NOT NULL THEN
          LET order_total = order_total + p_items[i].total_price
       END IF
    END FOR
    LET order_total = 1.1 * order_total
    DISPLAY order_total TO t_price ATTRIBUTE (GREEN)
END FUNCTION
FUNCTION item_total()
    DEFINE pa_curr, sc_curr INTEGER
    LET pa_curr = arr_curr()
    LET sc_curr = scr_line()
    LET p_items[pa_curr].total_price =
       p_items[pa_curr].quantity * p_items[pa_curr].unit_price
    DISPLAY p_items[pa_curr].total_price TO s_items[sc_curr].total_price
END FUNCTION

FUNCTION renum_items()
    DEFINE pa_curr, pa_total, sc_curr, sc_total, k INTEGER
    LET pa_curr = arr_curr()
    LET pa_total = arr_count()
    LET sc_curr = scr_line()
    LET sc_total = 4
    FOR k = pa_curr TO pa_total
       LET p_items[k].item_num = k
       IF sc_curr <= sc_total THEN
          DISPLAY k TO s_items[sc_curr].item_num
          LET sc_curr = sc_curr + 1
       END IF
    END FOR
END FUNCTION

FUNCTION insert_items()
    DEFINE idx INTEGER
```

```
   FOR idx=1  TO arr_count()
      IF p_items[idx].quantity != 0 THEN
         INSERT INTO items
            VALUES (p_items[idx].item_num, p_orders.order_num,
               p_items[idx].stock_num, p_items[idx].manu_code,
               p_items[idx].quantity, p_items[idx].total_price)
            IF status < 0 THEN
               RETURN (FALSE)
            END IF
      END IF
   END FOR
   RETURN (TRUE)
END FUNCTION

FUNCTION get_stock()
   DEFINE idx integer
   OPEN WINDOW stock_w AT 7, 3
      WITH FORM "stock_sel"
      ATTRIBUTE(BORDER, YELLOW)
   CALL set_count(stock_cnt)
   DISPLAY " Use cursor using F3, F4, and arrow keys; press ESC ",
      "to select a stock item" AT 1,1
   DISPLAY ARRAY p_stock TO s_stock.*
   LET idx = arr_curr()
   CLOSE WINDOW stock_w
   RETURN p_stock[idx].stock_num, p_stock[idx].manu_code,
      p_stock[idx].description, p_stock[idx].unit_price
END FUNCTION

FUNCTION get_item()
   DEFINE pa_curr, sc_curr INTEGER
   LET pa_curr = arr_curr()
   LET sc_curr = scr_line()
   SELECT description, unit_price
      INTO p_items[pa_curr].description,
         p_items[pa_curr].unit_price
      FROM stock
      WHERE stock.stock_num = p_items[pa_curr].stock_num
         AND stock.manu_code = p_items[pa_curr].manu_code
      IF status THEN
         LET p_items[pa_curr].description = NULL
         LET p_items[pa_curr].unit_price = NULL
      END IF
      DISPLAY p_items[pa_curr].description, p_items[pa_curr].unit_price
         TO s_items[sc_curr].description, s_items[sc_curr].unit_price
      IF p_items[pa_curr].quantity IS NOT NULL THEN
         CALL item_total()
      END IF
END FUNCTION

FUNCTION find_order()
   ERROR "Function not yet implemented"
   SLEEP 3
   RETURN
END FUNCTION
```

Form Specifications

This cust_order program uses two forms:

- The **orderform** form lets the user query for a customer and enter a new order for that customer.
- The **stock_sel** form is displayed within a window if the user needs to consult a stock list from the **orderform** form.

The orderform Form

```
DATABASE stores7

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      ]    PO Number:[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        ]
================================================================================
}

TABLES
customer orders items stock

ATTRIBUTES
f000 = customer.customer_num;
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 = "###-###-#### XXXXX";

f009 = orders.order_num;
f010 = orders.order_date, DEFAULT = TODAY;
f011 = orders.po_num;
f012 = orders.ship_instruct;
```

```
f013 = items.item_num, NOENTRY;
f014 = items.stock_num;
a1 = items.manu_code, UPSHIFT;
f015 = stock.description, NOENTRY;
f016 = items.quantity;
f017 = stock.unit_price, NOENTRY;
f018 = items.total_price, NOENTRY;
f019 = formonly.t_price TYPE MONEY;


INSTRUCTIONS
SCREEN RECORD s_items[4](items.item_num, items.stock_num, items.manu_code,
        stock.description, items.quantity, stock.unit_price, items.total_price)
```

The sel_stock Form

```
DATABASE stores7

SCREEN
{
   [f018][f019][f020           ][f021           ][f022     ][f023           ]
   [f018][f019][f020           ][f021           ][f022     ][f023           ]
   [f018][f019][f020           ][f021           ][f022     ][f023           ]
}

TABLES
stock

ATTRIBUTES
f018 = FORMONLY.stock_num;
f019 = FORMONLY.manu_code;
f020 = FORMONLY.manu_name;
f021 = FORMONLY.description;
f022 = FORMONLY.unit_price;
f023 = FORMONLY.unit_descr;

INSTRUCTIONS
DELIMITERS " "
SCREEN RECORD s_stock[3] (FORMONLY.stock_num THRU FORMONLY.unit_descr)
```

Module 1: globals.4gl

The **globals** file selects the **stores7** database and defines the following:

- **p_customer**, a program record with variables defined LIKE all columns of the **customer** table

  The **p_customer** record is used to store a customer row chosen from the results of a query by example.

- **p_orders**, a program record with variables defined LIKE selected columns from the **orders** table

  The **p_orders** record is used to store information such as order date, purchase order (P.O.) number, and shipping instructions during order entry.

- **p_items**, a program array of 10 records with variables defined LIKE selected columns from the **items** and **stock** tables

  The **p_items** array is used to store information such as stock number, quantity, and total price during order entry.

- **p_stock**, a program array of 15 records with variables defined LIKE selected columns from the **manufact** and **stock** tables

  The **p_stock** array is used to store a stock list that is displayed within the **stock_w** window on the screen form.

- **stock_cnt**, an INTEGER variable used as a counter

  The **stock_cnt** variable keeps track of the number of rows read from the database into the **p_stock** array.

Module 2: main.4gl

Module 2 contains the following program blocks and functions:

```
GLOBALS
MAIN
mess
clear_menu
fetch_stock
query_customer
```

The GLOBALS Statement

The GLOBAL statement establishes global variables from the **globals.4gl** file.

The MAIN Statement

The MAIN statement sets up the operating environment of the program as follows:

1. Begins with DEFER INTERRUPT

   If the user presses the **Interrupt** key with interrupts deferred, the global variable **int_flag** (defined by 4GL) is set to nonzero and program execution continues.
2. Opens and displays the **orderform** form
3. Displays the **ORDERS** menu, which provides the following options:
   - **Add-order** calls the **add_order** function, which allows the user to select a customer from the **customer** table, to enter an order for that customer on the **orderform** form, and to add the order to the database.
   - **Find-Order** is not currently implemented.
   - **Exit** exits the **ORDERS** menu and the **cust_order** program.

The mess Function

The **mess** function displays a message, sleeps for three seconds, and clears the message. The function accepts two arguments, displaying **str** (a character string) at **mrow** (a message line number).

The **query_customer** function assigns a value to **mrow**.

The clear_menu Function

The **clear_menu** function uses two DISPLAY statements to clear the first two screen lines.

The fetch_stock Function

The **fetch_stock** function is called from the **add_order** function if the user needs to look up stock information during order entry. The function retrieves information from the **stock** and **manufact** tables and stores it in the **p_stock** program array (displayed later by the **get_stock** function).

The **fetch_stock** function performs the following operations:

1. Declares **stock_list** as a cursor for a SELECT statement to retrieve the following information from the **stock** and **manufact** tables:
   - Stock number
   - Manufacturer code
   - Manufacturer name
   - Description
   - Unit price
   - Unit description

   The results are ordered by stock number.
2. Sets the global variable **stock_cnt** to `1`

   This variable serves as a counter during the subsequent FOREACH loop.

3. Uses a FOREACH loop to perform the query and to store the results in the **p_stock** program array

   This array contains the following global variables.

Variable	Defined LIKE
stock_num	stock.stock_num
manu_code	manufact.manu_code
manu_name	manufact.manu_name
description	stock.description
unit_price	stock.unit_price
unit_descr	stock.unit_descr

   The FOREACH loop works as follows:

   - For each row retrieved, the value of the counter **stock_cnt** increments by `1`. In this way, successively retrieved rows are stored in **p_stock[stock_cnt].***, where **stock_cnt** is a number from 1 to 15 (the maximum number of records the array can store).
   - The following IF test exits the FOREACH loop when the value of **stock_cnt** exceeds `15`:

```
IF stock_cnt > 15 THEN
EXIT FOREACH
END IF
```

4. Uses a LET statement following the FOREACH loop to decrement the value of the global variable **stock_cnt** by 1 to accurately reflect the number of records in the **p_stock** array

The query_customer Function

The **query_customer** function is called from the **add_order** function so the user can query for customer information. The function displays a submenu so the user can browse through the query results and choose a customer. The selected row is stored in **p_customer**, a program record with global variables defined LIKE *all* columns of the **customer** table.

Variable	Defined LIKE
customer_num	customer.customer_num
fname	customer.fname
lname	customer.lname
company	customer.company
address1	customer.address1
address2	customer.address2
city	customer.city
state	customer.state
zipcode	customer.zipcode
phone	customer.phone

The **query_customer** function accepts **mrow** (a message line number) as an argument. The **add_order** function passes the value of **mrow** to **query_customer**, which passes the value, in turn, to the **mess** function. The **query_customer** function returns one of the following values to the calling function (**add_order**) to indicate whether or not the user selects a customer:

- If the user does select a customer, **query_customer** returns the value `1` (TRUE).
- If the user does not select a customer or if no customer row satisfies the user's query, **query_customer** returns the value `0` (FALSE).
- If the user presses the **Interrupt** key to abort the query, **query_customer** returns NULL.

The **query_customer** function performs the following operations:

1. Defines local variables

Variable Name	Type
where_part	CHAR(200)
query_text	CHAR(250)
answer	CHAR(1)
mrow	SMALLINT
chosen	SMALLINT
exist	SMALLINT

   The values assigned to **chosen** and **exist** indicate whether the user's query retrieves at least one customer row (**exist** is TRUE) and whether the user selects a row from the query results (**chosen** is TRUE). Both **exist** and **chosen** must be TRUE in order for the **query_customer** function to return a TRUE value.

2. Clears the **orderform** form and the **ORDERS** menu
3. Displays a message that prompts the user to query for one or more customers
4. Uses a CONSTRUCT statement to allow the user to perform a query by example on the **customer** table:

```
CONSTRUCT where_part ON customer.* FROM customer.*
```

   The user can now enter selection criteria in one or more fields of the default **customer** screen record. When the user presses ESC, the selection criteria are stored in the local variable **where_part** (equivalent to the WHERE clause of a SELECT statement).

5. Uses an IF statement to test the value of the global variable **int_flag**

   With interrupts deferred, this flag is set to nonzero when the user presses the Interrupt key, and the program continues. If the interrupt test is true, the program performs as follows:

   a. Resets **int_flag** to FALSE to ensure that later interrupt tests are valid

   b. Clears the form and displays the message `Customer query aborted` on the error message line

   c. Sets the **p_customer.customer_num** global variable to NULL

      This clears any value retained from a previous query.

   d. Exits the function, returning a null value for **p_customer.customer_num**

6. Concatenates where_part to a character string that contains the rest of the SELECT statement and assigns the result to the **query_text** local variable

7. Prepares an executable statement called **statement_1** from the SELECT statement stored in **query_text**

   The statement selects all rows from the customer table that satisfy the user's selection criteria and orders the results by last name.

8. Declares **customer_set** as a SCROLL cursor for **statement_1**

9. Determines the rows that satisfy the query and leaves the cursor pointing before the first row of the active set

10. Retrieves the first row and assigns its values to the global program record **p_customer.***

11. Tests the value of the **status** global variable (defined by 4GL)

    If the query returned no rows, **status** equals `NOTFOUND`, and a LET statement sets the **exist** local variable to FALSE. If, however, the query returned at least one row, the program performs as follows:

    a. Sets the **exist** global variable to TRUE to indicate that the query returned one or more rows

    b. Displays the values in the **p_customer** program record in the corresponding fields of the screen form

    c. Displays the **BROWSE** menu

12. COMMAND clauses of the MENU statement establish the following options:

- **Next** displays the next customer row (if any) of the active set.
- **Previous** displays the previous customer row (if any) of the active set.
- **First** displays the first (and possibly the only) customer row of the active set.
- **Last** displays the last (and possibly the only) customer row of the active set.
- **Select** selects the displayed customer row, sets **chosen** to TRUE, and exits the menu.
- **Quit** sets **chosen** to FALSE and exits the menu without selecting a customer row.

Most of these options let the user browse through the active set. The **Next** option, for example, consists of the following program segment:

```
COMMAND "Next" "View the next customer in the list"
FETCH NEXT customer_set    INTO p_customer.*
   IF status = NOTFOUND THEN
     ERROR "No more customers in this direction"
           ATTRIBUTE(RED,REVERSE)
     FETCH LAST customer_set INTO p_customer.*
   END IF
DISPLAY BY NAME p_customer.* ATTRIBUTE(MAGENTA)
```

When the user selects the option, a FETCH statement attempts to retrieve the next row of the active set. If the value of the **status** variable indicates that the cursor has moved beyond the last row of the active set, a FETCH statement returns the cursor to the last row, and an ERROR statement displays an error message. A DISPLAY BY NAME statement displays the row retrieved by the appropriate FETCH statement on the screen. The **Previous**, **First**, and **Last** options work in approximately the same way.

13. Closes the cursor and calls **clear_menu** to clear the menu

    At this point, one of the following conditions exists:

    - The user's query returned no rows (**exist** is FALSE).
    - The user quit the **BROWSE** menu without selecting a customer (**exist** is TRUE and **chosen** is FALSE).
    - The user selected a customer from the **BROWSE** menu (**exist** and **chosen** are TRUE).

14. If the user's query returned no rows, the statement `IF NOT exist` is TRUE, and the program performs as follows:

    a. Clears the form

    b. Calls the **mess** function, which displays the message `No customer satisfies query` on the message line

    c. Sets the **p_customer.customer_num** global variable to NULL

    This clears any value retained from a previous query.

    d. Exits the function and returns FALSE to the **add_order** function

15. If the user quit the **BROWSE** menu without selecting a customer, the statement `IF NOT chosen` is TRUE, and the program performs as follows:

    a. Clears the form

    b. Calls the **mess** function, which displays `No selection made` on the message line

    c. Sets the **p_customer.customer_num** global variable to NULL

    This clears the value retained from the user's query.

    d. Exits the function and returns FALSE to the **add_order** function

16. If both of the preceding IF tests are FALSE, the user's query returned a row, and the user selected a row, in which case the **query_customer** function then returns TRUE to the calling **add_order** function

Module 3: order.4gl

Module 3 contains the following program block and functions:

```
GLOBALS
add_order
insert_order
order_total
item_total
renum_items
insert_items
get_stock
get_item
find_order
```

The GLOBALS Statement

The GLOBALS statement establishes global variables from the **globals.4gl** file and defines the following module variable.

Variable Name	Type
query_stat	INTEGER

The scope of a module variable is the module within which it is defined. This variable assumes the value returned by the **query_customer** function. As a module variable, it is available to any function of the **order.4gl** module.

The add_order Function

The **add_order** function lets the user enter an order into the database. During order entry, the function calls other functions to perform operations such as these:

- Select a customer from the **customer** table (the **query_customer** function).
- Display a stock list if the user wants to look up a stock number (the **fetch_stock** and **get_stock** functions).

- Renumber the items in the program and screen arrays should the user add or delete a row of the screen array (the **renum_items** function).
- Calculate and display an order total (the **order_total** function) as the user enters order items.

The **add_order** function performs the following operations:

1. Defines local variables

Variable Name	Type
pa_curr	INTEGER
s_curr	INTEGER

   The **pa_curr** and **s_curr** variables are used to store the values returned by the following built-in functions:

   - **arr_curr()** returns the value of the current program array row.
   - **scr_line()** returns the value of the current screen array row.

   The variables ensure corresponding values between arrays that might contain different numbers of rows.

2. Calls the **query_customer** function and assigns the returned value to the **query_stat** module variable

   The value is TRUE if the user selected a customer from the **BROWSE** menu. (See "The query_customer Function" on page C-27.)

   The call to **query_customer** also assigns a value to **mrow**. (See "The mess Function" on page C-25 and "The query_customer Function" on page C-27.)

3. Tests the value of **query_stat**. If **query_stat** is NULL or `0` (FALSE), the user has yet to select a customer, and a RETURN statement exits the function

4. If **query_stat** is TRUE, the **p_customer** program record contains the selected customer row, in which case a DISPLAY BY NAME statement displays the stored values in the corresponding fields of the screen form.

5. Displays the following message:

```
Enter the order date, PO number, and shipping
instructions.
```

6. Lets the user enter an order date, P.O. number, and shipping instructions on the **orderform** form:

```
INPUT BY NAME p_orders.order_date, p_orders.po_num,
        p_orders.ship_instruct
```

When the user presses ESC, the values from the screen fields are stored in the corresponding variables of the **p_orders** record.

7. Tests the value of the global variable **int_flag**

With interrupts deferred, this flag is set to nonzero when the user presses the Interrupt key, and the program continues. If the test is true, the function performs as follows:

a. Resets **int_flag** to FALSE to ensure that later interrupt tests are valid

b. Clears the form and displays `Order input aborted` on the error message line

c. Exits the function

8. Uses an INPUT ARRAY statement to assign values to the **p_items** program array from data in the **s_items** screen array

This screen array is defined in the INSTRUCTIONS section of the **orderform** specification to display the following information for up to four items:

- Item number
- Stock number
- Manufacturer code
- Description
- Quantity ordered
- Unit price
- Total price for that item

The user must enter a stock number, a manufacturer code, and an item quantity in the appropriate fields. The remaining fields are NOENTRY fields for which the program supplies values. Input ends when the user presses ESC to add the order or presses the **Interrupt** key to abort order entry.

The clauses of an INPUT ARRAY statement can execute in any order. They are discussed here in the order in which they appear:

- BEFORE FIELD

  When the user must enter a value, a BEFORE FIELD clause displays instructions such as these that follow:

  ```
  Press ESC to write order
  Enter a stock number or press CTRL-B to scan stock list
  ```

- ON KEY

  An ON KEY clause lets the user select an item from a stock list if the user presses CTRL-B from the **stock_num** or **manu_code** field.

  LET statements assign the number of the current program array row to the **pa_curr** local variable and the number of the current screen array row to the **s_curr** local variable. A call to **fetch_stock** retrieves a stock list from the database. (For details, see "The fetch_stock Function" on page C-25.)

  A call to **get_stock** opens a window and displays the stock list within the window (for details, see "The get_stock Function" on page C-43). When the user presses ESC to select an item from the list, **get_stock** returns selected values to the **add_order** function where they are assigned, in turn, to the current row of the **p_items** program array:

  ```
  CALL get_stock() RETURNING
    p_items[pa_curr].stock_num, p_items[pa_curr].manu.code,
    p_items[pa_curr].description,  p_items[pa_curr].unit  price
  ```

  DISPLAY statements display the selected stock number, manufacturer code, description, and unit price in the corresponding fields of the **s_items** screen array.

  A NEXT FIELD statement moves the cursor to the **quantity** screen field so the user can enter an item quantity.

- AFTER FIELD

  When the cursor moves out of the **stock_num** or **manu_code** screen field, an AFTER FIELD clause checks the values of the corresponding program array variables. If the variables are NOT NULL, the user has specified both a stock number and a manufacturer code. The program calls the **get_items** function to look up and display the description and unit price of the item.

  When the cursor moves out of the **quantity** screen field, an AFTER FIELD clause checks the value of the **unit_price** and **quantity** variables of the **p_items** program array.

  If the variables are NOT NULL, the user has specified a stock number, a manufacturer code, and a quantity. A call to **item_total** then multiples quantity by unit price and displays the result in the **total_price** screen field.

  If either variable is NULL, the user has yet to specify the required information. The program displays an error message and moves the cursor to the **stock_num** field (the first field).

- AFTER INSERT, DELETE

  The AFTER INSERT, DELETE clause executes if the user inserts or deletes a row of the screen array. In either case, the program renumbers the items in the program array and the screen array to ensure consecutive values in the two arrays (the **renum_items** function). The program then recalculates the displayed order total (the **order_total** function).

- AFTER ROW

  When the cursor moves to a new row of the screen array, the program calls the **order_total** function to calculate and display a total price for all items in the screen array.

  The INPUT ARRAY statement ends when the user presses ESC to add the order or presses the **Interrupt** key to cancel the order.

9. Tests the value of the **int_flag** global variable

   With interrupts deferred, this flag is set to nonzero when the user presses the **Interrupt** key, and the program continues. If the **interrupt** test is true, the program performs as follows:

   a. Resets **int_flag** to FALSE to ensure that later interrupt tests are valid

   b. Clears the form and displays the message `Order input aborted` on the error message line

   c. Exits the function

10. Calls the **insert_order** function to add the order to the database

The insert_order Function

The **insert_order** function is called from the **add_order** function to perform a transaction that adds an order to the database. Because one order can specify many items, an order consists of one row in the **orders** table and one or more rows in the **items** table.

The **insert_order** function performs the following operations:

1. Uses a WHENEVER ERROR CONTINUE statement to ensure that an error (**status** < 0) will not terminate the program
2. Begins the transaction (BEGIN WORK)
3. Inserts the following values into the **orders** table:

   ```
   0, p_orders.order_date, p_customer.customer_num,
      p_orders.ship_instruct,   p_orders.po-num
   ```

   In this list of values, `0` is a placeholder for the serial value 4GL will add to the **order_num** column of the table.
4. Checks the value of the **status** global variable to make sure that the INSERT INTO statement executed correctly

   If **status** is a negative number, an error has occurred, and the program performs as follows:

   a. Rolls back the transaction

      The database is as it was at BEGIN WORK.

   b. Displays an error message

   c. Exits the function

5. Assigns the serial number just added to the **orders** table (stored in SQLCA.SQLERRD[2]) to **p_orders.order_num** and displays the value on the screen
6. Calls **insert_items** to add the following information to the **items** table for each order item:
   - Item number
   - Order number
   - Stock number
   - Manufacturer code
   - Quantity
   - Total price

   If the **insert_items** function returns FALSE, the program performs as follows:
   a. Rolls back the transaction

      The database is as it was at BEGIN WORK.)
   b. Displays an error message
   c. Exits the function
7. Commits the modification to the database (COMMIT WORK)
8. Issues a WHENEVER ERROR STOP statement to override the earlier WHENEVER ERROR CONTINUE statement
9. Displays the message `Order added` on line 23 (**mrow**)
10. Clears the form

The order_total Function

The **order_total** function is called from the **add_order** function to add the values of the **total_price** column of the **p_items** array and to display the sum on the screen form.

The **order_total** function performs the following operations:

1. Defines local variables

Variable Name	Type
order_total	MONEY(8)
i	INTEGER

   The **i** variable assumes the value of the current row of the **p_items** program array. The **order_total** variable stores the sum of the values of **p_items[i].total_price**, where **i** identifies successive rows of the program array.

2. Assigns a starting value of `0.00` to **order_total**
3. Uses a FOR loop to call the **arr_count** function to determine the number of rows in the **p_items** program array and repeats the following sequence that number of times:

```
FOR i = 1 TO ARR_COUNT()
     IF p_items[i].total_price IS NOT NULL THEN
        LET order_total =
        order_total + p_items[i].total_price
     END IF
END FOR
```

   The value of **i** (initially `1`) increments with each repetition of the loop.

   The IF statement performs as follows:

   a. Checks that the value of **total_price** for the current row of the program array is not NULL.

      A null value plus any value is NULL.

   b. Adds the value (if it exists) to the value of **order_total** and assigns the sum to **order_total**

   The value of **order_total** therefore increases with each non-NULL row.

4. Adjusts the value of **order_total** for tax and shipping
5. Displays the adjusted order total in the **t_price** field of the screen array

The item_total Function

The **item_total** function is called from the **add_order** function or the **get_item** function to compute and display a total price for the current item (the current row of the **p_items** program array).

The **item_total** function performs the following operations:

1. Defines local variables

Variable Name	Type
pa_curr	INTEGER
sc_curr	INTEGER

2. Assigns the values returned by the following built-in functions to the **pa_curr** and **sc_curr** local variables:
   - **arr_curr()** returns the value of the current program array row.
   - **scr_line()** returns the value of the current screen array row.

   The variables are used to ensure corresponding values between the two arrays.
3. Multiplies unit price by quantity for the current item and assigns the result to **p_items[*pa_curr*].total_price**, where ***pa_curr*** is the current row of the program array
4. Displays the item total price in the **total_price** field of the appropriate screen array row

The renum_items Function

The **renum_items** function is called from the **add_order** function to renumber the items in the program array and the screen array if the user adds or deletes a row of the screen array.

The **renum_items** function performs the following operations:

1. Defines local variables

Variable Name	Type
pa_curr	INTEGER
pa_total	INTEGER
sc_curr	INTEGER
sc_total	INTEGER
k	INTEGER

2. Assigns the number of the current program array row to the variable **pa_curr**
3. Assigns the number of rows currently stored in the program array to the variable **pa_total**
4. Assigns the number of the current screen array row to the variable **sc_curr**
5. Assigns the total number of rows in the screen array to the variable **sc_total**
6. Uses a FOR loop to renumber the rows in the program array from the current row to the last row, as well as to renumber the rows in the screen array from the current row to the last displayed row

The insert_items Function

The **insert_items** function is called from the **insert_order** function to add rows to the **items** table. Each row contains an order number (**p_orders.order_num**) and values from the **p_items** program array. The **insert_items** function returns a value to the calling function to indicate whether the rows were successfully added.

The **insert_items** function performs the following operations:

1. Defines a local variable

Variable Name	Type
idx	INTEGER

   The **idx** variable assumes the value of the current row of the **p_items** program array.

2. Uses a FOR loop to call the **arr_count** function to determine the number of rows in the program array and repeats the following sequence that number of times:

```
FOR idx = 1 TO arr_count()
 IF p_items[idx].quantity !=  0 THEN
  INSERT INTO items
   VALUES (p_items[idx].item_num,  p_orders.order_num,
      p_items[idx].stock_num,    p_items[idx].manu_code,
      p_items[idx].quantity,     p_items[idx].total_price
  IF status < 0 THEN
   RETURN (FALSE)
  END IF
  END IF
END FOR
```

   The value of **i** (initially `1`) increments with each repetition of the loop.

The IF statement performs as follows:

a. Checks that the current row of the **p_items** program array contains a valid item quantity

   The user might have entered 0 by mistake.

b. Inserts a row into the **items** table provided there is a valid item quantity

c. Tests the value of the **status** global variable to make sure the INSERT statement executed correctly

d. If **status** is a negative number, an error occurred, and the **insert_items** function returns the value FALSE to the calling function.

3. The **insert_items** function returns the value TRUE to the calling function if the preceding FOR loop executes without error.

The get_stock Function

The **get_stock** function displays the stock list retrieved by the **fetch_stock** function within a window on the screen form. When the user presses ESC to select an item from the list, the function returns the item values in the current row of the **p_stock** program array.

The **get_stock** function performs the following operations:

1. Defines the **idx** local variable, to assume the value of the current row of the program array
2. Opens the **stock_w** window and displays the **stock_sel** form within the window
3. Calls the built-in function **set_count** to set the initial value of **arr_count()** to the number of rows currently stored in the program array

   This call is required before any DISPLAY ARRAY statement (step 5).
4. Displays instructions for selecting a stock item

5. Displays the values in the **p_stock** program array in the corresponding fields of the **s_stock** screen array

   At this point, the user can select an item as follows:

   - Press F3, F4, or the arrow keys to scroll through the rows in the program array.
   - Press ESC to select the current item.

6. Assigns the value returned by the **arr_curr** function to the **idx** local variable to identify the current row of the program array
7. Closes the **stock_w** window
8. Exits the function, returning the values of the selected row of the **p_stock** program array:

```
RETURN p_stock[idx].stock_num, p_stock[idx].manu_code,
   p_stock[idx].description, p_stock[idx].unit_price
```

The get_item Function

If the user enters a stock number and manufacturer code, the **get_item** function is called from the **add_order** function to look up and display a description and unit price for the corresponding item. If the user has also entered a quantity, the **get_item** function calls **item_total** to compute and display a total item price.

The **get_item** function performs the following operations:

1. Defines local variables

Variable Name	Type
pa_curr	INTEGER
sc_curr	INTEGER

2. Assigns the values returned by the following built-in functions to the **pa_curr** and **sc_curr** local variables:

   - **arr_curr()** returns the value of the current program array row.
   - **scr_line()** returns the value of the current screen array row.

   The variables are used to ensure corresponding values between the two arrays.

3. Retrieves the description and unit price of the current item from the **stock** table and stores the information in the **description** and **unit_price** variables of the **p_items** program array:

```
SELECT description, unit price
   INTO p_items[pa_curr].description,
        p_items[pa_curr].unit_price
   FROM stock
   WHERE stock.stock_num = p_items[pa_curr].stock_num
      AND stock.manu_code = p_items[pa_curr].manu_code
```

4. Checks the value of the **status** global variable to make sure the SELECT statement executed without error

   If an error occurred, the program assigns null values to the **description** and **unit_price** variables. (This clears any values assigned by the SELECT statement.)

5. Displays the values of **description** and **unit_price** for the current row of the **p_items** program array in the corresponding fields of the **s_items** screen array

6. Checks the value of **quantity** for the current row of the **p_items** array

   If the value is NOT NULL, the user has entered a quantity. The program therefore calls the **item_total** function to calculate and display a total item price

The find_order Function

The **find_order** function, intended to query the database for order information, is currently unimplemented. If the user chooses **Find-order** from the **ORDERS** menu, the function displays an error message.

Error Messages

An error message appears in the Command window if an error condition is detected by the Debugger. This section lists the Debugger error messages. It also describes how the Debugger responds to each error, and suggests corrective actions that you can take. If no system action is specified, the command was not executed.

Every Debugger error message is prefixed by a negative number. Use this *error number* to find the description of the error message in the pages that follow. Debugger error messages are listed in descending order, starting with -16300.

The error messages that appear in this section primarily refer to improper syntax in Debugger commands rather than to errors in your 4GL source code. If you see an error message number that is outside the range -16399 to -16300, you can refer to its listing in the ''Error Messages'' section near the end of the *INFORMIX-4GL Reference*.

If you are using a new version of INFORMIX-4GL or the Programmer's Environment, you might be advised of errors that were not flagged in earlier versions of 4GL.

Debugger Error Messages

-16300 **Description of error:** Identifier is too long.

System action: The command was not executed. If the error occurred during a READ command or while a **.4db** initialization file was being executed, other command lines of the input file might have been executed.

Corrective action: Identifiers cannot have more than 18 characters. See if you have omitted a separator between two identifiers, or select a new identifier of the appropriate length.

-16301 **Description of error:** A syntax error has occurred.

Corrective action: Check that you have not misspelled or omitted a keyword or identifier, included an extra command argument, or placed keywords out of sequence. See if you have omitted parentheses after a function name in a CALL command or included them in a VIEW command. You can enter `help all` to display a two-page synopsis of all the Debugger commands. Repeat the command, correcting any syntax errors.

-16302 **Description of error:** An illegal character has been found in the command.

Corrective action: Characters in Debugger commands are restricted to letters, numbers, blanks, underscores, and the special characters listed in the section "Conventions for Command Syntax Notation" on page 9-29. You might have pressed a key inadvertently, or you might have introduced illegal characters when you edited a file that provided input to a READ command.

If you create or modify a **.4db** file with a word processing program, be sure to invoke it in *nondocument* mode. Repeat the command, without including the illegal character (which might be a nonprintable control character).

-16303 **Description of error:** An illegal integer has been found in the command.

System action: The LET, PRINT, or Search command was not executed.

Corrective action: Your expression or search pattern includes a numeric string of more than 50 characters. Repeat the command, specifying a shorter expression or search pattern.

-16304 **Description of error:** An illegal floating point number has been found in the command.

System action: The LET, PRINT, or Search command was not executed.

Corrective action: Your expression or search pattern includes a floating point numeric string of more than 50 characters or includes more than one decimal point. Repeat the command, specifying a shorter expression or search pattern or fewer decimal points.

-16305 **Description of error:** Memory allocation failed.

Corrective action: Not enough system RAM is available to execute your command. Repeat the command at another time, when other users are making smaller demands on system RAM.

-16306 **Description of error:** Found a quote for which there is no matching quote.

Corrective action: You might have omitted a quotation mark from a name or string, or included an extraneous quotation mark. Repeat the command, using an even number of single (') or double (") quotes.

-16307 **Description of error:** Quoted string is too long.

Corrective action: The maximum length of a quoted string is 256 characters. Reduce the string to a length within this limit.

-16308 **Description of error:** Missing function name.

System action: The CALL command was not executed.

Corrective action: You must specify the name of a function in a CALL command. Repeat the command, specifying the name of a function, followed by left and right parentheses ( ). If the function requires arguments, include the list of arguments within the parentheses, separating multiple arguments by commas.

-16309 **Description of error:** Internal buffer limit exceeded.

Corrective action: Your command has too many characters. A Debugger command can include no more than 256 characters.

If you are specifying an ALIAS, BREAK, or TRACE command that includes many command lines within braces, you might consider using nested aliases. This procedure can specify a function key or a short string as the equivalent of hundreds of keystrokes, by specifying it as the alias of a list of aliases that each represents fewer than 256 characters. It is easier to use READ commands, rather than aliases, to enter multiple commands.

-16310 **Description of error:** Keyword expected.

Corrective action: You have omitted a required command option. You can enter `help all` to display the names of all the Debugger commands, or read Chapter 9, "The Debugger Commands," for information about specific Debugger commands. Repeat the command, supplying a valid keyword and any other required specifications.

-16311 **Description of error:** Command [*name*] is not recognized.

Corrective action: You have misspelled the name of the command or improperly abbreviated it. If what appears after the $ prompt in your command window seems to be a valid command keyword, you probably also pressed a nonprinting key. You can enter `help` to display the names of all the Debugger commands. Repeat the command, using a valid form of the keyword.

-16312 **Description of error:** Missing or misplaced = sign.

System action: The ALIAS or LET command was not executed.

Corrective action: The LET command always requires an equal sign (=), as does ALIAS (unless you specify the * option). Repeat the command, making sure that you use the equal sign in the appropriate place.

-16314 **Description of error:** Missing filename.

System action: The Debugger was not loaded, or the DUMP, PRINT, READ, TRACE, or VARIABLE command was not executed.

Corrective action: If the Debugger is already loaded, you have not specified a filename after a READ command or after a command to redirect output to a file. Repeat the command, specifying the name of an input or output file.

If you were at the system prompt, you have used the **-f** command-line option to invoke the Debugger, but you have not specified the name of an initialization file. Repeat the command, but this time either omit the **-f** or specify the name of a **.4db** initialization file after the **-f**.

-16315 **Description of error:** Missing).

Corrective action: You have either omitted a right parenthesis from a command argument, or you have included an extraneous left parenthesis. Repeat the command, making sure to include an even number of parentheses.

-16316 **Description of error:** A small integer is expected.

System action: The GROW command was not executed.

Corrective action: You must supply a positive or negative integer as an argument of a GROW command. Repeat the command, specifying the number of lines to be added to the size of the window. The sum of this integer and the current size must be in the range from 1 to (*L* - 6), where *L* is the number of lines that your terminal can display (usually 24).

-16317 **Description of error:** Program variable name expected.

System action: The LET command was not executed.

Corrective action: A LET command requires three arguments:

- The identifier of a program variable
- An equal sign (=)
- An expression whose value is assigned to the variable

You omitted the first argument. Repeat the command, supplying appropriate arguments.

-16318 **Description of error:** Positive number or name expected.

System action: The ENABLE, DISABLE, NOBREAK, or NOTRACE command was not executed.

Corrective action: The ENABLE, DISABLE, NOBREAK, and NOTRACE commands all require an argument. This can be a breakpoint or tracepoint name, reference number, function name, or the ALL option. If the argument that you entered in the Command window looks valid, you might have also pressed a nonprinting character. Repeat the command, supplying an appropriate argument.

-16321 **Description of error:** Alias [*name*] is an existing command keyword.

System action: The ALIAS command was not executed.

Corrective action: The name of an alias cannot be the complete form of a Debugger command keyword. (You are allowed, however, to assign an abbreviated form of a command keyword as an alias.) Repeat the command, this time specifying a different alias.

-16322 **Description of error:** Missing directory list.

System action: The Debugger was not loaded.

Corrective action: You have used the **-I** command-line option to invoke the Debugger, but you have not specified a source file search path. Repeat the command, but this time either omit the **-I** or specify the name of a search path after the **-I**.

-16324 **Description of error:** User cannot specify more than one filename.

System action: The Debugger was not loaded.

Corrective action: You can only use the **-f** option to specify one initialization file in a command line. Repeat the command but with no more than one **-f** *filename* specification, where *filename* is the name of a **.4db** file. After you have invoked the Debugger, you can use READ commands to execute additional **.4db** files.

-16325 **Description of error:** Missing or extra parameter.

System action: The Debugger was not loaded.

Corrective action: You have omitted or repeated the **-I** or **-f** symbols in a command line. Repeat the command, following the syntax of the **fgldb** command (described in "Invoking the Debugger" on page 8-11).

-16326 **Description of error:** Missing program name.

System action: The Debugger was not loaded.

Corrective action: If you invoke the Debugger at the system prompt, you must include as an argument the name of a compiled 4GL program (or else the symbols **-V**). Repeat the command, this time specifying the name of a compiled 4GL program.

-16327 **Description of error:** Cannot locate file [*filename*].

System action: The READ or VIEW command was not executed, or the Source window might be empty.

Corrective action: You might have omitted or misspelled the filename of a **.4db** input file (either in a READ command or after the **-f** option of a command line) or the filename or pathname of a **.4gl** source file in a VIEW command or in a command line.

If the input file is not in your current directory, you must prefix its name with a pathname or add its directory to the source file search path with a USE command. Then repeat the command, supplying a valid file specification.

-16329 **Description of error:** Pathname too long.

Corrective action: The Debugger will accept pathnames of up to 70 total characters. Repeat the command, specifying a shorter pathname.

-16330 **Description of error:** Cannot open file [*filename*] for reading.

System action: The READ or VIEW command was not executed, or the Debugger was invoked without executing the commands in the **.4db** file that you specified after the **-f** option of a command line.

Corrective action: Make sure that you specified the correct filename in your READ or VIEW command, or after the **-f** option when you invoked the Debugger at the system prompt. If the name was correct, the file might be damaged or read protected. If you do not have permission to read it, refer to the discussion of access privileges in your operating system documentation, or contact your system administrator.

-16331 **Description of error:** Too many recursive aliases.

System action: The ALIAS command was not executed.

Corrective action: In some debugging tasks, aliases that reference other aliases can save time or circumvent limits on the number of characters in a Debugger command line. For example, you can specify **alias1** to be a substring of **alias2**, which can be a substring of **alias3** and so forth up to **alias5**.

You cannot, however, specify more than five levels of aliases between your keystrokes and the fully expanded commands that they alias. Repeat the command, using fewer levels of aliases.

-16334 **Description of error:** Internal error—null keyword encountered.

Corrective action: Please notify the Informix Technical Support Department.

-16335 **Description of error:** Window can not be adjusted by number of lines specified.

System action: The GROW command was not executed.

Corrective action: The minimum number of lines in the Source window or in the Command window is 1, not counting the Source window line that displays the current module name. On a standard 24-line terminal, the maximum number of lines in either window is 18 or (*L*-6) for nonstandard terminals with *L* lines.

The argument of a GROW command is not the new window size but the *increment* to the current window size. If you want to change the size of a window in the Debugger screen, enter a GROW command to produce window sizes within this range.

-16337 **Description of error:** No previous search pattern.

System action: The Search command was not executed.

Corrective action: You cannot enter a Search command without specifying a search pattern, unless earlier in the same debugging session you specified a search pattern. Repeat the command, specifying a valid search pattern.

-16338 **Description of error:** Cannot continue execution.

System action: The STEP or CONTINUE command was not executed.

Corrective action: You cannot invoke CONTINUE or STEP commands unless a 4GL program has begun (but not terminated) execution. If you have not begun execution, or if your application has terminated normally or by a fatal error, use CALL or RUN. After execution begins, you must then suspend execution by a breakpoint or by an Interrupt command before you can invoke CONTINUE or STEP. See also the section "Active Functions and Variables" on page 9-20.

-16339 **Description of error:** Break or trace name [*name*] is not unique.

System action: The BREAK or TRACE command was not executed.

Corrective action: A name that you assign to a breakpoint or tracepoint in a BREAK or TRACE command cannot duplicate the name of any existing reference point (including disabled breakpoints or tracepoints). This message will also appear after a READ command that sets a named breakpoint or tracepoint, if you read the same input file twice. Repeat the command, substituting a unique name or no name.

-16340 **Description of error:** Invalid backslash encountered.

Corrective action: Your command includes a backslash symbol that cannot be interpreted as a command-line continuation symbol. Repeat the command without invalid backslashes.

-16341 **Description of error:** Line number [*line-no*] not in specified module.

System action: The BREAK or TRACE command was not executed.

Corrective action: Your BREAK or TRACE command cannot specify a line number that is greater than the line number of the last executable statement in the specified module. If you specified no module name, the line number cannot be greater than the last executable statement in your current module. You can use a VIEW command to display a 4GL source module, and then enter `$` to determine the number of lines in the module.

Check to see if you have specified the correct module and line number. Repeat the command, specifying a valid line number or module name.

-16342 **Description of error:** Internal error—cannot set breakpoint.

System action: The BREAK command was not executed.

Corrective action: Please notify the Informix Technical Support Department.

-16343 **Description of error:** Invalid module name [*name*] specified.

System action: The BREAK, TRACE, or VIEW command was not executed.

Corrective action: You have specified a module or function that is not part of the current 4GL program. Check to see if you have misspelled the name or if you are confusing the names of modules or functions from different programs. Repeat the command, specifying a valid module name or function name.

-16344 **Description of error:** Invalid function name [*name*] specified.

System action: The BREAK or TRACE command was not executed.

Corrective action: You have specified a function that is not part of the current 4GL program. Check to see if you have misspelled the name, or if you are confusing the names of functions from different programs, or if you neglected to compile and concatenate a program module that contains the function. Repeat the command, specifying the name of a function in the current 4GL program.

-16345 **Description of error:** Cannot set breakpoint in a 4GL library function or user C function.

System action: The BREAK command was not executed.

Corrective action: You can only set a breakpoint at a 4GL function, not at a C function or ESQL/C function. If you want to suspend program execution when a C function is called, set a breakpoint by specifying the line number of the 4GL statement that calls the C function, rather than specifying the name of the function. See Appendix B, "Calling C Functions," for more information about C functions.

-16346 **Description of error:** Cannot open output file [*filename*].

System action: The DUMP, PRINT, TRACE, VARIABLE, or WRITE command was not executed.

Corrective action: See if the output file already exists but is damaged or write-protected. See if you have permission to write in the specified directory (or in the current directory if you specified no pathname). If you do not have write permission to create or modify the output file, refer to the discussion of access privileges in your operating system documentation, or contact your system administrator.

If you already have write permission in the current directory, make sure that the output filename that you supplied does not end in a slash symbol that your operating system interprets as a pathname, rather than a filename. Repeat the command, specifying a valid name for your output file or specifying a pathname to a directory where you have write permission.

-16347 **Description of error:** Invalid breakpoint or tracepoint number [*reference number*] specified.

System action: The DISABLE, ENABLE, NOBREAK, or NOTRACE command was not executed.

Corrective action: You cannot specify a reference number in an ENABLE, DISABLE, NOBREAK, or NOTRACE command with a zero or negative value or with a value for which no corresponding breakpoint or tracepoint currently exists. (You can enter `list break trace` to display your current reference numbers.) Repeat the command, using a valid reference number.

-16348 **Description of error:** Breakpoint or tracepoint [*name*] is not active.

System action: The DISABLE command was not executed.

Corrective action: You cannot use a DISABLE command to deactivate a breakpoint or tracepoint that is already disabled. No action is necessary, unless you had meant to ENABLE a breakpoint or tracepoint, or to DISABLE a different point. Enter `list break trace` to display your current breakpoints and tracepoints.

-16349 **Description of error:** Use NOBREAK for breakpoints and NOTRACE for tracepoints.

System action: The NOBREAK or NOTRACE command was not executed.

Corrective action: Do not specify the name or reference number of a tracepoint in a NOBREAK command, and do not specify the name or reference number of a breakpoint in a NOTRACE command. Invoke the appropriate command.

-16350 **Description of error:** Breakpoint/tracepoint already disabled/enabled or can not be determined.

System action: The DISABLE or ENABLE command was not executed.

Corrective action: The Debugger cannot find any corresponding breakpoints or tracepoints on which to carry out your ENABLE or DISABLE command. Enter `list break trace` to display all your current reference points.

-16351 **Description of error:** Variable [*name*] could not be located.

Corrective action: You cannot reference a variable that appears only in a C function. For 4GL variables, make sure that you correctly entered the variable name. If the variable is not in the current function or module, you must qualify its name. See the "Scope of Reference" on page 9-16 for more information.

-16352 **Description of error:** File [*filename*] has been modified. (**.4gl** is newer than **.4go**)

System action: The command was executed, but line numbers listed in the Source window might not correspond to the statements of the compiled program.

Corrective action: If you change the source code in a **.4gl** file, you must use the p-code compiler (at the system prompt or from within the Programmer's Environment) to recompile a corresponding **.4go** file. If your program includes several modules, you must also concatenate the compiled modules before you can use the Debugger on the program. Check to see if files with both **.4go** and **.4gi** extensions of the same filename exist. Even if your **.4go** file is the appropriate recompiled version, the Debugger will attempt to interpret the **.4gi** version first.

-16353 **Description of error:** Executable command is invalid when executing trace-point commands.

System action: The TRACE command was executed, but the embedded CALL, CONTINUE, RUN, or STEP command was not.

Corrective action: The list of command lines in your TRACE command includes a CALL, CONTINUE, RUN, or STEP command. The Debugger does not allow any of these four program execution commands to appear in the command list of a TRACE command. Repeat the TRACE command, excluding from its command list any invalid program execution command.

-16354 **Description of error:** Function [*name*] not found.

System action: The CALL command was not executed.

Corrective action: You have specified a function that is not part of the current 4GL program. Check to see if you have misspelled the name, or if you are confusing the names of functions from different programs. Make sure that your 4GL source code specifies and calls the function that you want to execute. Repeat the command, specifying a valid function name.

-16355 **Description of error:** Function [*name*] requires parameters.

System action: The CALL command was not executed.

Corrective action: The required argument of a CALL command is a function name, followed by parentheses, with a list of any arguments within the parentheses. If you are not sure what parameters should be passed to the function, you can use a VIEW command to display its source code. Repeat the command, specifying appropriate arguments.

-16356 **Description of error:** Too many parameters passed to function [*name*].

System action: The CALL command was not executed.

Corrective action: Your argument list in a CALL command includes too many parameters. If you are not sure how many parameters should be passed to the function, you can use a VIEW command to display the source code of the function. Repeat the command, specifying the appropriate number of arguments in parentheses after the function name.

-16358 **Description of error:** Variable name expected.

System action: The LET command was not executed.

Corrective action: A LET command requires as its first argument a variable name. Do not put quotes around a variable name. Repeat the command, specifying the name of a program variable.

-16359 **Description of error:** Cannot assign values to records; assignments must be to record members.

System action: The LET command was not executed.

Corrective action: A LET command can assign a value to a member of a record but not to the whole record. Repeat the command, specifying the name of a variable within the record.

-16360 **Description of error:** Cannot assign values to arrays; assignments must be to array elements.

System action: The LET command was not executed.

Corrective action: A LET command can assign a value to an element of an array but not to the whole array. Repeat the command, specifying an element within the array. Enter `help let` to see an example.

-16362 **Description of error:** No current function.

System action: The DUMP command was not executed.

Corrective action: There is no current 4GL function if you have not yet invoked a RUN or CALL command. Even after RUN or CALL, there is no current function if execution terminated normally rather than being stopped by a breakpoint, by an **Interrupt** or STEP command, or by a fatal error. Use RUN or CALL to begin or restart program execution.

-16363 **Description of error:** Variable in function [*name*] is not active.

System action: The CALL, LET, or PRINT command was not executed.

Corrective action: You have referenced a variable whose value is assigned by a function that has not yet been called or by a function that has already returned. A LET or PRINT command can only reference active variables. A variable named as an argument of a CALL command must be a global variable or a nonglobal variable in an active function (the functions listed by a WHERE command).

If a CALL command requires arguments, you must substitute a constant for the name of any variable that is neither active nor global. See the section "Active Functions and Variables" on page 9-20.

-16364 **Description of error:** Unknown option [*name*].

System action: The TURN command was not executed.

Corrective action: The Debugger cannot identify an argument of your command. You probably used an invalid option or abbreviation. Repeat the command, specifying a valid option. Entering `help turn` displays the options of TURN.

-16365 **Description of error:** Breakpoint or tracepoint [*name*] is already active.

System action: The ENABLE command was not executed.

Corrective action: The ENABLE command has no effect on breakpoints or tracepoints that are already enabled. Check whether you have specified the correct name or reference number. If these are correct, enter `list break trace` or search the command buffer to see if a prior DISABLE command deactivated the wrong breakpoint or tracepoint. If necessary, invoke appropriate ENABLE or DISABLE commands.

-16366 **Description of error:** Error occurred while trying to write to a file.

System action: Not all the output from the command was written to the output file.

Corrective action: Either you experienced a hardware error with your hard disk, or your file system is full. Use the Escape feature to display the remaining storage capacity of your current drive, and contact your system administrator.

-16367 **Description of error:** Need to specify a specific record member or array element.

Corrective action: You cannot use the name of an array as an argument of a CALL command. You cannot set a breakpoint or tracepoint on an array or record, or include the name of a record or of an array, in an expression. Repeat the command, specifying a member of the record or an element in the array.

-16368 **Description of error:** Expression contains variables from different functions.

System action: The BREAK, LET, or PRINT command was not executed.

Corrective action: Do not use variables from multiple functions in the same expression. If you need to know the value of such an expression, you must use PRINT to evaluate the variables from each function separately. Then enter these values as constants in the BREAK, LET, or PRINT command, rather than the names of the variables.

-16369 **Description of error:** Cannot initialize application device [*device-name*].

System action: The APPLICATION DEVICE command was not executed.

Corrective action: You must specify the name of another terminal that has the same **termcap** or **terminfo** entries as the terminal from which you invoked the Debugger. The second terminal must be logged in by your account name. Repeat the command, specifying a valid device.

-16370 **Description of error:** No application device specified.

System action: The APPLICATION DEVICE command was not executed.

Corrective action: You must specify the name of another terminal that has the same **termcap** or **terminfo** entries as the terminal from which you invoked the Debugger. Enter `tty` from the terminal that you want to use as your application device. Its screen will display its terminal pathname.

Repeat the APPLICATION DEVICE command, specifying an appropriate terminal device name. The device name must not be the name of the device from which you invoked the Debugger.

-16371 **Description of error:** Read file not specified.

System action: The READ command was not executed.

Corrective action: You must specify the name of a **.4db** file in a READ command. Repeat the command, specifying a filename.

-16372 **Description of error:** Variable or expression expected.

System action: The PRINT command was not executed.

Corrective action: You must identify what you want the Debugger to display in a PRINT command. Repeat the command, specifying the name of a program variable, record, array, or expression.

-16373 **Description of error:** Cannot print or make assignments before execution has started.

System action: The DUMP, LET, or PRINT command was not executed.

Corrective action: You cannot use a LET command to assign a value to a variable or a DUMP or PRINT command to display information about a variable or function until after execution starts. The same restriction applies after execution terminates normally or after a CLEANUP command.

Invoke the RUN or CALL command to begin execution, and then repeat the command. (It might be necessary to set a breakpoint or press the **Interrupt** key to prevent normal termination.)

-16374 **Description of error:** Break command contains IF without having a condition specified.

System action: The BREAK command was not executed.

Corrective action: The IF keyword in a BREAK command must be followed by an expression. The breakpoint has no effect while the condition is FALSE (0). Repeat the command without IF or with IF and a condition.

-16375 **Description of error:** Cannot view C-library function [*name*].

System action: The VIEW command was not executed.

Corrective action: The VIEW command cannot display the source code of C functions or ESQL/C functions. To examine C source code, you must use the Escape feature to invoke an operating system command that displays the C source file.

-16376 **Description of error:** Break or trace name [*name*] does not begin with an alpha character.

System action: The BREAK or TRACE command was not executed.

Corrective action: The first character in the name of a breakpoint or tracepoint must be a letter. The subsequent characters can be letters, numbers, or underscores (_). The name must be enclosed between single (') or double (") quotes. Repeat the command, specifying a valid name and enclosing the name in single or double quotes.

-16377 **Description of error:** Cannot retrieve values of global variables before execution begins.

System action: The PRINT command was not executed.

Corrective action: A PRINT command cannot display the value of a global program variable until after execution commences. Invoke the RUN or CALL command to begin execution, and then repeat the PRINT command. In this situation, you can specify constants or the names of global variables as arguments of CALL. (It might be necessary to set a breakpoint or to press the **Interrupt** key to suspend program execution before normal termination.)

-16378 **Description of error:** A small positive integer is expected.

System action: The TIMEDELAY command was not executed.

Corrective action: You cannot enter a TIMEDELAY command without an argument or with a negative number as the argument. Repeat the command, specifying zero or a positive integer as the number of seconds delay in the Source window or Command window.

-16381 **Description of error:** Cannot set breakpoint or tracepoint—no current module.

System action: The BREAK or TRACE command was not executed.

Corrective action: You cannot set a breakpoint or tracepoint without referencing a module or function unless there is a program module in the Source window. You have probably ignored an error message that appeared when you were unable to load a 4GL source file.

-16382 **Description of error:** Command file [*filename*] is currently being processed.

System action: A READ command invoked either by a READ command or by an initialization file was not executed.

Corrective action: You have used a READ command, nested in an initialization file or in the input file of a READ command, that refers to one of the following:

- To itself
- To a previous READ command input file
- To a **.4db** initialization file that has not yet completed executing all of its commands

Avoid nesting READ commands, or make sure that no **.4db** file contains a READ command that creates an infinite loop.

-16383 **Description of error:** Number of nested READ commands limit exceeded.

System action: A nested READ command was not executed.

Corrective action: You invoked a READ command that invokes another READ command, that invokes another, and so forth for more than 10 nested READ commands. You cannot nest more than 10 READ commands. Simplify your **.4db** files.

-16384 **Description of error:** Line in **.4db** file exceeds maximum length.

System action: The READ command was not executed.

Corrective action: You cannot have more than 256 characters in a single Debugger command line. If you are using semicolons as command separators, break the line instead into separate commands, and repeat the commands.

You should avoid Debugger commands that have more characters in a single line than your screen or list device can display. Use the backslash continuation symbol to divide long command lines into shorter segments, or use aliases.

-16385 **Description of error:** Call to function [*name*] failed.

System action: The CALL command was not executed.

Corrective action: The logic in your 4GL or C language function might be defective or might not support the argument list that you specified in a CALL command. Use the VIEW command to examine the source code of a 4GL function. Use the Escape feature to examine the source file of a C function.

-16386 **Description of error:** Search string exceeds maximum length.

System action: The Search command was not executed.

Corrective action: The maximum length of a search pattern specification whose first character is a quote (") is 80 characters, or a maximum length of 50 after any other first character. See if you unintentionally pressed a Search command key (? or /). Repeat the Search command but specify a shorter pattern.

-16387 **Description of error:** Program is not currently being executed.

System action: The WHERE command was not executed.

Corrective action: A WHERE command cannot display your active functions until after execution commences. Invoke the RUN or CALL command to begin execution, and then repeat the command. (It might first be necessary to set a breakpoint to suspend execution before normal termination.) See the section "Active Functions and Variables" on page 9-20.

-16388 **Description of error:** Cannot create Debugger window.

Corrective action: You are probably out of memory. Repeat the command at another time, when other users are making smaller demands on system RAM.

-16389 **Description of error:** Filename exceeds maximum length.

Corrective action: A filename cannot exceed 80 characters. Repeat the command, specifying a shorter filename.

-16390 **Description of error:** Error occurred while reading file [*filename*].

Corrective action: The Debugger encountered an error while trying to read a 4GL source file. Check to make sure that the file still exists and that it is not corrupted.

-16391 **Description of error:** Internal error—attempt to highlight invalid line number.

Corrective action: Repeat the command. If the same error message is repeated, please notify the Informix Technical Support Department.

-16392 **Description of error:** No database name specified.

System action: The DATABASE command was not executed.

Corrective action: The DATABASE command requires as its argument the name of a database. Repeat the command, specifying the name of a database that can be accessed by the Debugger.

-16393 **Description of error:** Expression or variable contains invalid substring.

System action: The BREAK, LET, or PRINT command was not executed.

Corrective action: The Debugger cannot interpret an expression or variable in your command. To specify substrings of a character string, you must enter the name of the character variable, followed by two numbers, separated by commas, and enclosed within a pair of square brackets, as in the expression:

```
charstring[n1,n2]
```

Here **charstring** must be of type CHAR(*n*), where $n > 1$, and **n1** and **n2** must have positive integer values between 1 and *n* inclusive.

This error message appears if **n2** is larger than *n*, or if **n1** is greater than **n2**. Repeat the command, supplying valid specifications of the substring.

-16394 **Description of error:** Cannot access the help messages.

System action: The HELP command was not executed.

Corrective action: The Debugger help messages are in a file called **fgldb.iem** that the INSTALL program copies to directory **INFORMIXDIR/msg**. This file has been damaged, deleted, or read protected, or is unavailable to you for some other reason. Ask your system administrator to restore your access to this file.

Index

A

B

C

D

E

F

G

H

I

J

K

L

Q

R

S

T

U

V

W

X

Z

Symbols