Ingres 10.0

Forms-based Application Development Tools User Guide

INGRES

ING-10-FAD-02

Contents

PART 1: General Information

Chapter 1: Introduction

This section contains the following topics:

The Forms-based Application Development Tools User Guide describes, in one online guide, how to use either Vision or Applications-By-Forms (ABF) to develop forms-based applications that access Ingres databases. It also describes how to use embedded-forms programming to develop forms-based applications without the use of either ABF or Vision. Finally, it describes how to use the Ingres Fourth Generation Language (4GL) to customize applications.

Note: For information on developing applications with the GUI tool, OpenROAD, see the *OpenROAD Workbench User Guide* and other OpenROAD guides.

This guide is divided into the following parts:

Part 1—General Information

Introduces the tools and languages covered in this guide, presents general terms and concepts, and explains how to plan and create applications.

Part 2—Vision

Consists of nine chapters and four appendixes that explain how to use Vision to quickly develop applications without writing programming code and how to access Vision functions through Applications-By-Forms.

Part 3—Applications-By-Forms

Consists of three chapters and four appendixes that describe how to use ABF to create the frames and procedures in a forms-based application. ABF is more difficult to use than Vision, but gives you more control over the application.

Part 4—Embedded Forms Programming

Consists of three chapters that describe how to create applications that accept and display information on user-defined forms, without the use of either ABF or Vision. (The forms must have been created using the Visual Forms Editor. For instructions about creating forms, see *Character-based Querying and Reporting Tools User Guide*.) One appendix provides EQUEL/FORMS examples.

Part 5—4GL Reference

Consists of six chapters and five appendixes that provide a complete reference to 4GL statements and procedures. 4GL is used with both Vision and ABF to develop and customize applications.

Note: This part of the guide covers basic database manipulation statements. For information about SQL and descriptions of other SQL statements, see the *SQL Reference Guide*. For commands used only with a distributed database, see the *Database Administrator Guide*.

Part 6—Common Appendixes

Lists the menu operations available in the Vision and ABF main windows and discusses sharing a Vision or ABF installation with other developers.

The initial chapters in each part provide introductory information for novice users; the remaining chapters in each part provide more detail. The appendixes in each part contain information specific to the tool or language described in that part.

Common appendix information is provided at the end of the guide.

What You Must Know

This guide contains reference information appropriate to all users of Vision, ABF, embedded-forms programming, and 4GL. Before using these tools and languages you must:

- Understand the basic concepts of Ingres, such as databases and tables
- Know how to use Ingres forms and menus
- Be familiar with Visual Forms Editor

Special Considerations

This section discusses issues you must understand before using Vision, ABF, or embedded-forms programming.

Enterprise Access product compatibility

If you are working through Enterprise Access products, see your Enterprise Access documentation for information about command syntax that can differ from that described in this guide. Refer also to the *OpenSQL Reference Guide* for query language details.

4GL fully supports all commands and extensions to access databases through Ingres Federated Database Support and the Enterprise Access products.

Query Languages

The industry standard query language, SQL, is used as the standard query language throughout the body of this guide.

Some of the text and examples in this guide pertain only to Ingres query language, Ingres SQL.

Ingres is compliant with ISO Entry SQL92. In addition, numerous vendor extensions are included. For details about the settings required to operate in compliance with ISO Entry SQL92, see the *SQL Reference Guide*.

For a description of the QUEL statements available and information on using ABF or 4GL with QUEL, see the *QUEL Reference Guide*.

Terminology

This guide observes the following distinctions in terminology:

- A *command* is an operation that you execute at the operating system level. An extended operation invoked by a command is often referred to as a utility.
- A *statement* is an operation that you embed within a program or that you execute interactively from a terminal monitor. A statement can be written in a database query language (SQL or QUEL), Ingres 4GL, a host programming language (such as C), or in the Report-Writer language (for report specifications).

Chapter 2: Overview of Tools and Languages

This section contains the following topics:

This chapter introduces common terms and concepts. It also describes how the application development tools and languages are used together to plan and create applications.

How the Tools and 4GL Work Together

You can use either Vision or ABF to develop forms-based applications that access Ingres databases. Using these tools, you can define, test, and run fully developed applications without using a programming language. You can also use 4GL to take advantage of the advanced features of Vision or to provide specialized processing in ABF applications.

The following sections summarize the relationship between the tools and languages. For specific information about one of the tools or languages, see the appropriate part of this guide.

Vision and 4GL

Vision automatically generates the 4GL code that runs your applications. This 4GL code is based on standard SQL. (Vision does not support QUEL-based 4GL.) If necessary, you can edit the 4GL code for a specific type of Vision frame called a User frame.

You can add functions to the Vision frames by writing 4GL statements and expressions called *escape code*. You can also modify the Vision template files or add your own 4GL code to perform specific functions.

ABF and 4GL

You can develop forms-based applications that access Ingres databases with ABF without using 4GL. For many purposes, Query-By-Forms (QBF) frames and report frames are sufficient. However, for customized frames (such as the start frame), for specialized processing, and for procedures, you must write 4GL code.

4GL is the language for implementing menu operations and other activations. It allows you to control the user's movement among the frames and procedures of an ABF application. It allows you to associate specific operations with each frame to control the operation of the form. Because the form is the input and output medium for a forms-based application, a large part of any forms-based application involves operations that get data from, and display data on, the form. 4GL makes these interactions easy to specify.

Use 4GL to:

- Define custom processing steps for the application.
- Design the overall flow of an application in a series of consistent, easy-to-use menus.
- Fine tune an application by indicating what happens when the application user chooses a menu operation, presses a key, or tries to leave a particular field.
- Specify operations to query and update data in the database.
- Perform selective processing on table fields.
- Perform conditional processing, such as movement from field to field depending on the application user's data entry.
- Call other frames, 4GL procedures, Third Generation Language (3GL) procedures, Ingres tools, or the operating system.
- Pass the result of a query or calculation to another frame.
- Use local variables for calculation or data that the application user does not see.
- Carry out multi-row queries with submenus.
- Create entry forms.
- Perform functions such as clearing the screen, displaying error messages, or causing the terminal to beep.
- Allow for runtime data input.

Forms Runtime System

All of the tools and languages in this guide are designed for use on *character-based* terminals. Therefore, they all make use of the Forms Runtime System (FRS) to display a graphical-looking form on non-graphical, character-based terminals.

FRS was originally designed for use in UNIX or VAX midrange environments with VT-style ASCII terminals. When you use FRS, an environment variable, term_ingres, defines the type of character-based terminal you are using and a *termcap* file provides the hardware-to-software mapping FRS uses to determine which key was pressed. The standard termcap file delivered with Ingres supports over 500 varieties of character-based terminals and can be easily customized to support others.

If FRS applications are run on a graphical platform (like Microsoft Windows), they must run inside a character terminal emulation window.

Vision, ABF, and FRS

If you develop an application using Vision or ABF, the key mappings are defined by FRS. For example, the keys mapped to the FRS Upline and Downline commands allow you to highlight the row you want. FRS applications look like ABF and Vision applications.

FRS and 4GL

In 4GL, you use a key activation operation to control what happens when the application user presses a specific key on the keyboard. The operation is specified not for a particular function or control key, but for an FRS command, which is then mapped to a function or control key. 4GL provides the inquire_forms statement to retrieve information at runtime from FRS on the key that caused the activation. It also provides the set_forms statement to set various features of FRS dynamically.

4GL statements can activate 3GL or host language procedures that include embedded SQL statements and embedded forms statements.

4GL lets you assign a null value to a nullable variable through FRS.

4GL and Query Languages

4GL statements are based on Ingres SQL, industry standard SQL, and the Ingres proprietary QUEL. The statements available to you in a 4GL program include most SQL statements as well as Ingres SQL extensions. In most cases, the query statements in 4GL correspond very closely to standard SQL releases.

One of the strengths of 4GL lies in incorporating the power of ANSI standard SQL and embedded SQL, reducing the need to call separate embedded language procedures from your application. Using only 4GL, you can specify queries with exactly the same syntax as with SQL.

4GL does not include embedded language FORMS features that are infrequently used, such as bare table fields. To use them, call an embedded SQL (ESQL) procedure.

In addition to embedded language procedures, 4GL statements can also activate 4GL procedures and SQL or database procedures. See Procedures (see page 45) for more information.

- For a full description of the features and syntax of a query language or embedded language, see the appropriate Ingres reference guide.
- For lists of reserved words used in 4GL, SQL, QUEL, and other query languages, see Keywords (see page 1317).

Terms and Concepts

This section defines terms and concepts you must know, and describes the components of an application.

Relational Database Management System

Before you can begin to develop applications that access Ingres databases, you must understand some of the basic concepts and functions of Ingres.

Ingres is called a Relational Database Management System (RDBMS) because of the way it handles information. If you break this term up into its individual parts, it becomes much easier to understand:

- A *database* is a set of related information that Ingres stores for you in a computer. For example, your company has an inventory database of products and orders and a personnel database with information on its employees.

  The basic units of information that a database stores are called *data*.
- *Management* means that Ingres takes care of storing your data for you. It lets you put data into your databases—to add new customers, for example—or retrieve data from it—to produce a report of all items sold last month, for example.
- Ingres is a *system* because you can combine its different components to organize and access your data in the way that works best for you. Ingres has a variety of tools for such functions as creating reports, designing forms to make data entry easier, and retrieving data from a database.
- Finally, Ingres is *relational* because it stores your data in tables that make it easy for you to see how the various pieces of information relate to each other. These tables are illustrated in the next section.

Tables

An Ingres table contains *columns* for each type of data and *rows* with all the information for each entry in the table. The following table shows data about customers:

Name	Address	City	Balance
Armand Karol	1000 Pine St.	San Francisco	$9,500
LaDette Watson	666 Rose Ave.	Berkeley	$1,250
Betsy Stark	33 Schuman Dr.	Castro Valley	$4,375
My Lin Truong	100 Sheila Ave.	Oakland	$5,500

Each item in the top row (for example, Name) is the title of a column. Each row under it provides information about one customer.

The user of an application usually needs to access one or more tables to look up information or manipulate the data (that is, add, delete, or change the data). A typical application displays table information and allows the user to choose the kind of data manipulation required.

Database Queries

To look up information in Ingres tables, questions called *queries* are used. Queries retrieve specific data. For example, you can ask for the price and quantity of all parts in inventory or produce a list of all orders filled this month. You can display the answer to the query on a terminal or print it out as a report.

Adding, deleting, or changing data in a database is also a query, because you are asking the database to accept the new data.

Applications

The data in a database is only useful when it can be accessed quickly and easily. An *application* provides an efficient and flexible way to access data.

An application is a program that solves a problem. It is an interface that provides end users with the information they want from the database and the tools they use to manipulate the data.

- The information is provided by a set of queries that access data to meet specific needs.
- The tools are provided by menus that can be used to add, delete, or change data. Menu items can also access other Ingres tools to run reports.

An application accesses a particular database. Although all users of the application are accessing the same database, different people use the application in different ways to solve different parts of the problem. For example, assume a company's customer orders are stored in a database. An order clerk can use an application to enter a new order into the database, while a supply manager might use it to check on inventory amounts. An application can also allow a single user to access related information in different ways to perform a variety of tasks.

Each application has an *owne*r, a *creation date*, and a *modification date*. The *owner* is the person who creates the application.

Components

Applications are made up of the application program itself and a set of *components*, all of which you define. These can include the following:

Component	Description
Frames	The basic operational units of applications. The user interacts with the application through forms and menus on frames.
Procedures	Separate modules that perform specific operations. These are composed using 4GL, a host language code, or SQL.
Tables	Database tables containing data on which the application operates.
Reports	Data formatted for display or printing.
Variables	Contain the data that the application manipulates.
Constants	Named values global in scope for the application.
Record Types	Named groupings of data that can be treated as a single component. Record types can be grouped into arrays.

Following a description of their naming conventions, many of these components are described in detail later in this chapter.

Naming Conventions for Applications and Components

Applications and their components must follow these naming conventions:

- Names of the following objects can be 256 bytes long: Table, column, partition, procedure, procedure parameter, rule, sequence, synonym, constraint.

  Names of the following objects can be 32 bytes long: Database, owner, user, group, profile, role, schema, location, event, alram, node, Forms, JoinDefs, QBFnames, Graphs, Reports.

  Frame names and source files (*.osq) must be unique in the first 8 characters. Many programs, such as compilers, require the path to the source and object files.

  Windows: Limit names to 8 to 12 characters maximum to avoid exceeding the 126-character Microsoft Windows command line limit.

- The special characters _, #, $ and @ can be included in component names, except the character # cannot be included in procedure names.
- Component names must begin with an alphabetic character (A through Z) or underscore (_).

- Component names are not case sensitive.
- Table names cannot begin with "ii". These names are reserved for use by Ingres.
- Ingres database components—such as tables and views—can be specified using delimited identifiers. Delimited identifiers are surrounded by double quotes. Using delimited identifiers allows you to use special characters in object names, or to use keywords that are not otherwise legal as object names. Ingres tools components—such as applications and frames—cannot be specified with delimited identifiers. For more information on delimited identifiers, see the *SQL Reference Guide*.

Enterprise Access Products

For databases accessed through Enterprise Access products, component naming conventions are identical to the above. In particular:

- Names can be 1 to 32 bytes long.

  Windows: Limit names to 8 to 12 characters maximum to avoid exceeding the 126-character Microsoft Windows command line limit.
- Most names are case insensitive, regardless of the case sensitivity of the underlying database accessed through an Enterprise Access Product. Table names and some other database names can be case sensitive if the underlying DBMS supports this.

The following sections describe the application components in the order in which you encounter them as you design an application.

Frames

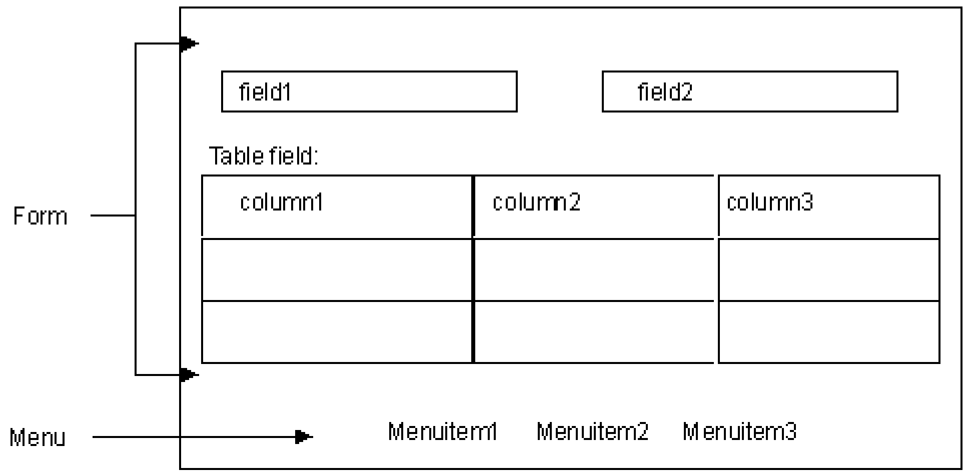
Each Vision or ABF application is a collection of related units called *frames*. Each frame generally corresponds to one computer window of information. It is through the frame that the user communicates with Ingres and manipulates the data in the database.

The application user sees the application as a sequence of frames, although all users cannot see the same set of frames. For example, a salesperson would probably not want to see the data that an order entry clerk needs.

A frame has two main visual components (the parts that appear on the display):

- The form determines how the frame's data is displayed. Forms are described in more detail in the next section.
- The *menu* at the bottom of the frame gives the user a choice of operations to perform on the frame. The user can query the database, run reports, use Ingres tools, invoke operating system utilities, and perform other tasks.

The following figure illustrates the components of a frame:



Forms

The *form* is what the user sees. It can be used just as you would use a form on a piece of paper, such as an order blank.

Forms contain fields where the user can enter information into the database or display information from the database. The cursor shows the location of activity, as the user goes from field to field, filling in or viewing data.

Forms are created automatically when you use Vision to develop applications. In ABF, one of the steps in creating a frame is using the ABF FormEdit operation to create a form. Default forms are also provided.

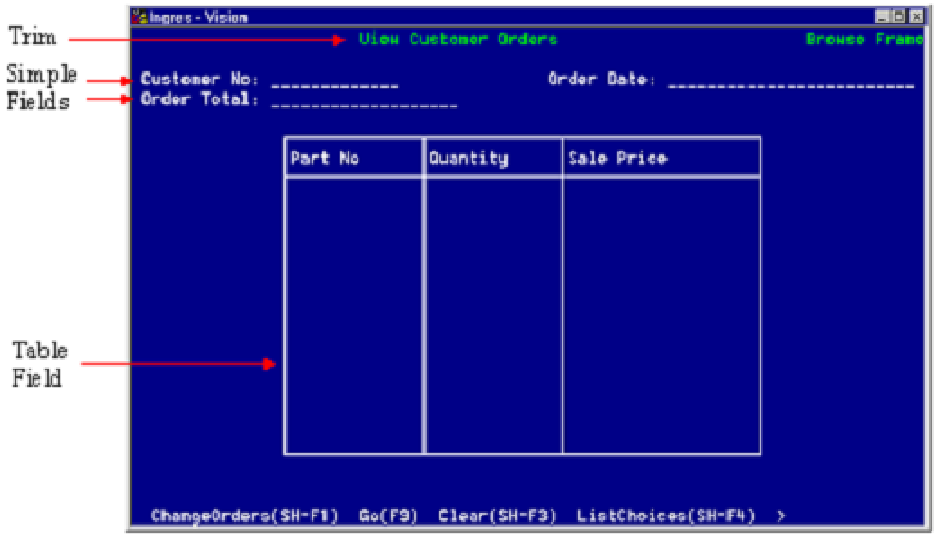
When you edit a form in either ABF or Vision, Visual-Forms-Editor (VIFRED) is called. VIFRED lets you design and change the layout of a form, define validation criteria (or edit checks) and error messages, and specify the way the form is displayed in the window. (For more information on VIFRED, see *Character-based Querying and Reporting Tools User Guide*.)

- You can use the same form in more than one frame to provide consistency in the way data is displayed, as long as no frame is nested below another frame that uses the same form. If you create duplicate forms with different names, there is no restriction on their use in frames.
- You can use the same form in different applications that use the same database, as long as the source code for the form appears in the source code directory of each application that uses it.
- You can copy forms from one database to another to use the same form in applications that use different databases.

Ingres forms have two visual components:

- Trim
- Fields

The following figure illustrates the components of a form:



Trim

Trim includes:

- Any instructions or general information on the form
- Lines, boxes or other visual enhancements to the form

You use trim to make the form look better or easier to use. Trim does not affect how the form handles data or how it communicates with Ingres.

Fields

Fields are areas on a form that display data. The data in a field is called the field's value. The columns in the table being queried determine the fields that appear on the form. (If the query changes, a new form is generated to reflect the changes.)

A form can have two types of fields:

- *Simple fields* display one value at a time, with the title of the column and the value in one row displayed next to each other across the window
- *Table fields* display multiple records arranged in horizontal rows and vertical columns

Procedures

In addition to frames, your application can include procedures. A *procedure* is a set of command statements that are called by name from an application to perform calculations or processing.

Procedures encapsulate operations that you want to use in an application, but which are not provided by Ingres tools, such as Query-By-Forms (QBF). You must write procedures in 4GL, 3GL, or SQL.

After you write a procedure, you can call it from a frame or another procedure. When called, a procedure executes sets of statements and returns data to the calling frame.

For more information on creating procedures with ABF, see Building Applications (see page 351). For more information about creating procedures with Vision, see Defining Frames without Visual Queries (see page 159).

Synonyms and Views

You can create a frame that refers to a *synonym* or a *view*, instead of a table name:

- A synonym is an alternate name for a table.
- A view is a virtual table.

All discussion about tables in this guide applies to synonyms and views, except that synonyms and views cannot be created with the ABF Tables Utility, as described in Building Applications (see page 351). You use query language statements (4GL, SQL, or QUEL) to create synonyms and views. For more information, see your query language guide.

Data Types

Ingres tables consist of columns of data. The *data type* of the column indicates the type of information that is contained in the column and how Ingres handles the information. Variables, constants, and procedures that return values also use data types to indicate what type of data is accepted.

Vision and ABF both support the standard Ingres data types, with the exception of long varchar, long bit, byte, byte varying, and long byte.

Note: If a Vision frame is based on a table that has a column of one of those types, that column is not displayed or included in the query. The remaining columns in the table are displayed.

In addition to using standard Ingres data types, you can define record types and arrays. Record types and arrays (special kinds of data types) are discussed in detail in Record Types (see page 48).

For more information about data types, see the *SQL Reference Guide*.

Variables

Variables contain the data that the application manipulates. This includes data that is displayed to the user as well as data that is used internally in frames and procedures.

Variables are created using ABF frames or 4GL code. The user can change the value that the variable represents as the application runs.

There are three types of variables in 4GL:

Simple Variables

Are single data items that contains one value, of any Ingres data type.

Record Variables or Records

Are a named set of attributes. Each record variable must be of a defined record type. See Record Types (see page 48).

Arrays

Are named sets of records. See the section Arrays in Record Types (see page 48).

Variables can have global or local scope in an application:

Global Variables

Provide data for an entire application. They can be used to maintain a value that is set once and rarely changed, such as the user name. They can be referenced in any frame or procedure, allowing easy access to the information. A global variable can be any data type.

Local Variables

Provide data for a specific procedure, frame, or form. They can be used only within the procedure or frame for which they are declared. A local variable can be any data type, and is defined with 4GL code.

Global variables can be used in any statement in which a local variable can be used. A local variable overrides a global variable with the same name.

You can declare a local variable to be based on an existing form, table field, or table. Global variables cannot be based on a form object.

Constants

Constants are numeric or text literals that have names. They are global in scope for an application. You can use them to organize applications by giving a common name to identical objects, or for program documentation.

You set the name and value in advance using ABF frames. You can change a set of constant names, such as those of menu items, to fit different contexts or to appear in another language. You cannot assign a new value to a constant while the application is running.

Constants can be used in any statement where a local variable can be used, except on the left side of assignment statements. However, a local variable overrides a constant with the same name.

Arrays or records cannot be made up of constants.

Constants can be of any simple data type, such as numeric or string, but cannot be of any complex data type, such as date or money.

Record Types

The *record type* is a data type that you use to treat several different pieces of data as one unit, for example, name and phone number. Each of these units is called a *variable of record type*. Each piece of data is called an *attribute*. An attribute can be a simple data type, another record type, or an array. A data value or a variable for the record type is called a *record*.

To define a record type in ABF or Vision, list its attributes. Record types defined in ABF or Vision can be local or global in scope.

To define a record type in 4GL, declare it to be the type of an existing form, table field, or table, or of an ABF-declared record type. Record types defined in 4GL are local in scope.

After you define a record type, you can declare a variable as that record type. This is a *record variable*. (See Variables above.) A record variable is declared and used the same way you declare variables of simple Ingres data types, such as integer or char.

You can use a record type to create any number of variables with the same attributes. You can also use the record type to define an array.

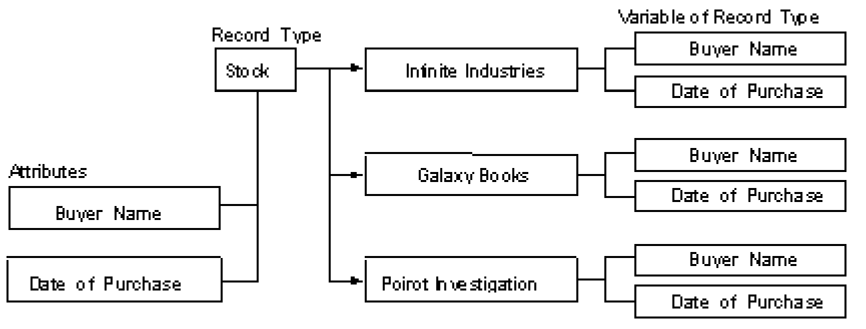
Records

Records allow you to organize your application by grouping data logically. They can help provide consistency throughout your application. (Records are similar to "records" in Pascal and "structures" in C.)

- You can operate on an entire record as a single unit by using its name in a statement.
- You can perform operations on the values in the record by using its name and the names of its attributes in a statement.
- You can assemble records in arrays (see Arrays below).

For example, you can retrieve a record from a database table and assign all values from the database record to a single 4GL record. You can then map the record as a set of fields in a display (like a row in a table), or you can pass it as a unit to a frame or 4GL procedure. You can also work with the individual attributes of the record with a notation combining the record and attribute names. But the attributes of a record are not ordered, and cannot be referenced by index unless they are records in an array.

The following figure shows the structure of a record type and records. The record type is Stock. The records are named after the imaginary companies, Infinite Industries, Galaxy Books, and Poirot Investigation. The attributes of the record type are the two fields of Stock: Buyer_Name and Date_of_Purchase.



Galaxy Books is a local or global variable of record type Stock.

Arrays

An array is a collection of records. Like records, arrays allow you to group data logically and create consistency throughout your application. You can define arrays as global variables through ABF frames or as local variables through 4GL declarations.

Like the arrays defined in 3GL languages, 4GL arrays consist of records that are all of the same record type. By referencing the array itself, you can work with the entire set of included records as a unit.

Unlike 3GL arrays, 4GL arrays are dynamic, allowing you to assign any number of records. The array automatically adjusts as you insert and delete components at any index position.

The index is an integer tag that distinguishes the records in an array from each other. Any record in an array can be addressed by its index at any time.

4GL supports only arrays of records; it does not support arrays of simple data types, as the C programming language does. While you cannot have an array of integers or characters, you can create records from these data types and then make an array of those records. For example, an array can be of a record type Employee, which is made up of three simple fields: Name, Emp_No, and Social_Security_No.

How to Develop an Application

The steps for developing an application are described in this section.

Steps

These are the general steps in the application development process:

1. Plan the application.
2. Create the application.
3. Define the frames and any other components in the application. (For ABF, this includes global components and procedures.)
4. Test the application during and after the definition stage.
5. Create an executable image of the application when testing is complete.
6. Run the executable image from the operating system.

The first two steps are described in the rest of this chapter. Steps that work differently because you are using Vision or ABF are described in detail in the Vision and ABF parts of this guide.

How to Plan the Application

Take time to plan an application carefully before you create it. The following guidelines can help ensure that your application is useful and effective:

- Design the flow of the application as a whole, including the hierarchy of its parts; that is, decide which frames call other frames, procedures, or other Ingres tools. (The application flow diagram in Vision lets you see the structure of your application as you build it, but you must not use it as a substitute for planning the application in advance.)
- Decide on a unique name for the application.
- Understand the users of your application and the information they need. Based on that, decide whether to restrict access to certain parts of the application. For example, your application can contain salary information that only managers are allowed to see. Define different frames as separate points of entry for different groups of users and decide where to use passwords.
- Be sure that the users of your application have the necessary privileges to use the tables that the application accesses. For example, if your application includes a frame to change records in a table, users must have update privileges for that table.

  See the database administrator for your database if you need more information about user privileges.

- Consider where you can utilize:
  - Global variables, where data is provided for the entire application
  - Records, where a group of values is logically related
  - Constants, where a set value can stand for a value, number, formula, or quoted string
- Design the individual frames and procedures in the application to give quick access to the information the users need.
- Decide which frame to use as the *start* frame. The start frame or procedure (sometimes called the top frame) is the point at which a user starts the application.
  - You can establish more than one starting point, allowing users to enter an application at exactly the frame they need without having to move through a sequence of menus.
  - In ABF, you can use the Application Defaults frame to define a default start frame or procedure.
- Know your database by knowing what the tables are and what data they contain, so that you know which tables to use with which frames. Also, be aware of any tables you must create for an application.

  You can examine the structures of existing tables at various points while creating an application (see Examining Tables (see page 245)). It is generally more efficient to have the tables you need before you create an application, but if necessary you can define new tables:
  - In Vision, use the Tables Utility to create new tables while working in Vision.
  - In ABF, you can define new tables within the database as you create an application. To enter data into newly created tables, call QBF. These procedures are described in Building Applications (see page 351).
- For large applications, decide in what order to build your application; that is, whether to work horizontally, creating all the peer frames at each level, or vertically, creating a child of the top frame and its entire tree before moving on to the next child of the top frame.

  One advantage of creating an entire tree at a time is that you can test each tree after you create it.
- Decide where to use advanced features to better meet your users' needs.

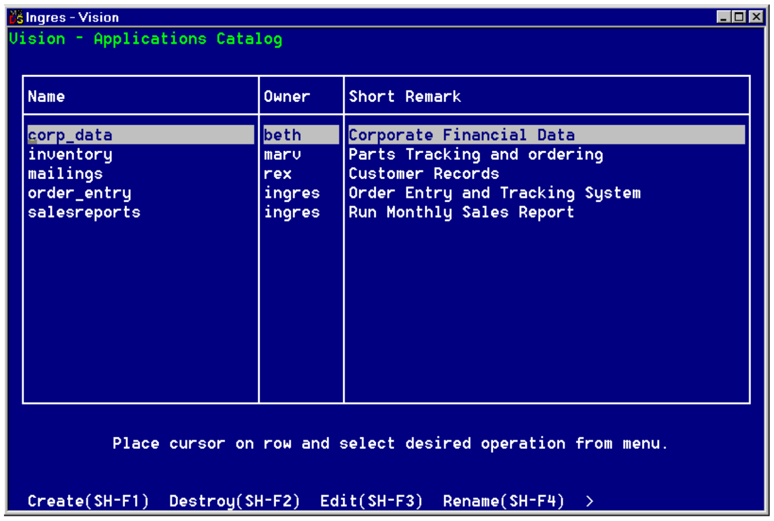
  These advanced features let you specify a wide range of complex queries. In general, however, you must try to keep your applications as simple as possible to make them easier to use and maintain.

How to Create the Application

The steps for creating an application are described in this section.

The Applications Catalog Window

The Applications Catalog window is the first window you see when you start Vision or ABF without specifying an application. Here is an example of an Applications Catalog window:



```
Vision - Applications Catalog

Name              Owner    Short Remark

corp_data         beth     Corporate Financial Data
inventory         marv     Parts Tracking and ordering
mailings          rex      Customer Records
order_entry       ingres   Order Entry and Tracking System
salesreports      ingres   Run Monthly Sales Report

       Place cursor on row and select desired operation from menu.

 Create(SH-F1)  Destroy(SH-F2)  Edit(SH-F3)  Rename(SH-F4)  >
```

The Applications Catalog contains:

- The name of each application in the database (you identify the database when you start Vision or ABF)
- The user ID of the person creating the application—either the database administrator (DBA) or a specific user
- A description of the application

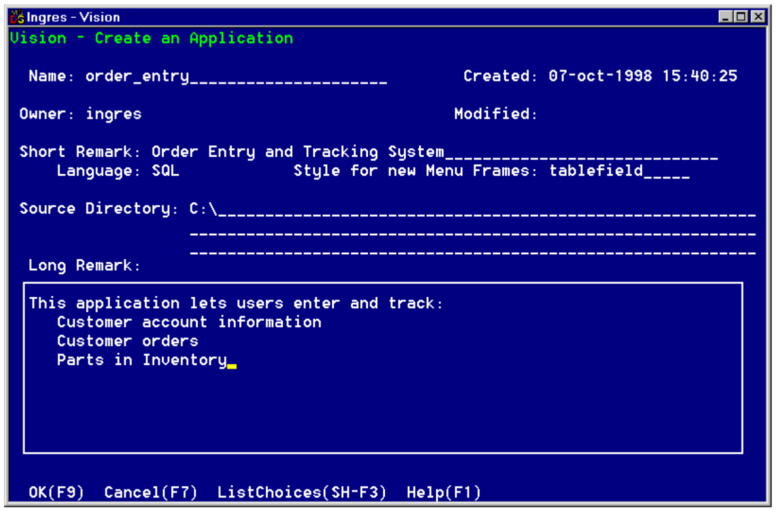
The Applications Catalog is a *display-only* window; that is, you cannot change any of the entries directly in the window.

You create and manage your applications through the menu operations at the bottom of the Applications Catalog window. For example, you can use the Rename operation to rename an application or the MoreInfo operation (described in The MoreInfo about an Application Window section) to change the description. For a list of the menu operations for all the windows in this chapter, see Menu Operations (see page 1375).

The Create an Application Window

To create a new application

1. Select Create from the menu operations at the bottom of the Applications Catalog window. The Create an Application window is displayed:



```
Vision - Create an Application

  Name: order_entry_____________________        Created: 07-oct-1998 15:40:25

 Owner: ingres                                 Modified:

 Short Remark: Order Entry and Tracking System_____________________________
     Language: SQL            Style for new Menu Frames: tablefield_____

 Source Directory: C:\_________________________________________________________
                   ____________________________________________________________
                   ____________________________________________________________
  Long Remark:

  This application lets users enter and track:
     Customer account information
     Customer orders
     Parts in Inventory_

  OK(F9)  Cancel(F7)  ListChoices(SH-F3)  Help(F1)
```

   You use this window to provide basic information about the application.

2. Type the application name in the Name field. This name must not be the name of any other application that accesses this database and must follow the standard object naming conventions (see Naming Conventions).

3. Press Tab. The cursor skips the Created, Owner, and Modified fields, because this information is filled in automatically. The owner is the user ID of the person creating the application. The Modified date field remains blank until the application is changed.

4. Enter a one-line description of the application in the Short Remark field. This description is optional, but it appears on the Applications Catalog window so is useful in identifying the application.

5. Press Tab.

   - If you are using Vision, the cursor skips the Language field, and the value SQL is entered to indicate that Vision uses the Structured Query Language to send queries to the database.

   - If you are using ABF, enter the query language for the complete application: SQL or QUEL. (Do not attempt to include two query languages within one application.) SQL is the default, but you can change this if your installation supports only QUEL or if your installation supports both, but you prefer QUEL.

6. For Vision only, the cursor moves to the Style for New Menu Frames field. Take one of the following actions:
   - Press Tab to accept the default that all menu frames in this application be displayed as table fields.
   - Enter single-line (or use the ListChoices operation to select this item) to specify that all menu frames be created with single-line menus.

   You can override the application-wide default for specific menu frames that you create. For more information, see Creating Frames (see page 87).
7. The cursor is now positioned on the Source Directory field. This field displays the full pathname of the directory that holds the source code for your application. The default is the directory in which you started Vision or AFB, but you can change it here by typing over the current value. If you do specify a new source directory, be sure to include the full pathname.

   For ABF, see Directories for Source Code and Application Components (see page 349).
8. Press Tab. The cursor moves to the Long Remark field. Use it to enter an optional description of up to eight lines about the application.
9. When you have finished entering your information, select OK from the menu. This displays the window you must use to define the frames in the application.

For more information on the menu items on this window, see Menu Operations (see page 1375).

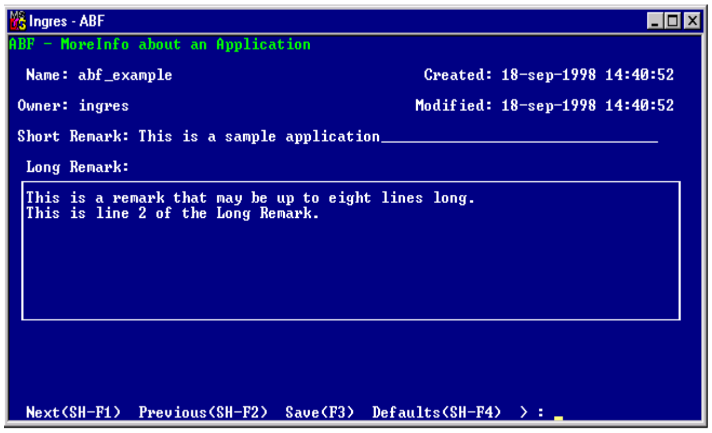
The MoreInfo about an Application Window

After you create an application, you can use the MoreInfo About an Application Window to view the information you entered on the Create an Application window. Also, you can add a Short or Long Remark or edit the ones you entered.

To display the MoreInfo About an Application window:

1. Position the cursor on the name of an application on the Applications Catalog window.
2. Select MoreInfo.

   The MoreInfo About an Application window is displayed:



```
Ingres - ABF
ABF - MoreInfo about an Application

  Name: abf_example                                Created: 18-sep-1998 14:40:52
 Owner: ingres                                    Modified: 18-sep-1998 14:40:52
 Short Remark: This is a sample application________________________________
  Long Remark:
  This is a remark that may be up to eight lines long.
  This is line 2 of the Long Remark.

  Next(SH-F1)  Previous(SH-F2)  Save(F3)  Defaults(SH-F4)  > : _
```

   This window shows you the information you entered when you created the application (or that you previously entered in this window).

   The Modified field now contains the last date on which you modified the MoreInfo About an Application window.

3. To add or change the Short or Long Remark:
   a. Enter new text or type over the current text in the Short Remark field.
   b. Tab to the Long Remark field.
   c. Enter a new Long Remark or type over the current one.
   d. Select Save to save any changes.
4. Select End to return to the Applications Catalog window.

The menu items on this window are described in Menu Operations (see page 1375).

PART 2: Vision

Chapter 3: Overview of Vision

This section contains the following topics:

Vision is a tool for developing forms-based applications. It provides you with an active working environment, including a visual display of your application, as well as prompts and menu operations that give instructions for building your application.

Knowledge of a programming language is not needed. Vision automatically generates the programming code that runs your applications. To take advantage of some of Vision's advanced features, however, you must be able to write statements in Ingres 4GL.

Note: Even if you are an experienced application developer, you can use Vision to build and test new applications quickly and customize the code that Vision generates to make your applications more extensive or flexible. You can also include additional code that you have written yourself.

Before you begin to use Vision to create and define frames, complete the following tasks:

- Read the section on frame types.
- Look at the examples of Vision-generated frames.
- For a discussion of the operations that users can perform on each of the Vision-generated frame types, see Vision Applications from a User's Perspective (see page 297).

These steps can help you decide which frame types to use in your applications.

How to Enhance Your Applications

Vision lets you take an adaptive approach toward developing applications. You can begin by using Vision to develop simple applications. As your information needs and your ability to use Vision and Ingres tools increase, you can gradually enhance your applications.

Develop your applications in these stages:

- Create an application using the frames, forms, and queries that Vision generates.
- Customize your queries with Vision's Visual Query Editor.
- Modify the forms that Vision creates.
- Specify more complex queries by using Vision's advanced features and including 4GL statements.
- Add frames that work with other Ingres tools, such as Report-By-Forms.
- Modify the code that Vision generates for a specific frame.
- Include User frames and procedures.

These development stages are discussed throughout this guide. Depending on your familiarity with Vision and the other Ingres tools, you can use as few or as many of them as you need to create applications with Vision.

Note: It is important to test your application frequently while you were building it.

For more information about the VIFRED, see the *Character-based Querying and Reporting Tools User Guide.*

Components of a Vision Application

Use Vision by assembling various components into an application. The main components of an application are:

- Frames
- Visual queries

The sections that follow describe these two main components.

Frames

An application is a collection of related units called *frames*. Each frame generally corresponds to one computer window of information.

The application user sees the application as a sequence of frames. It is through the frame that he or she communicates with Ingres and manipulates the data in the database. (Frames are described in more detail in Overview of Tools and Languages (see page 35).)

Frames consist of forms and menus.

Forms

The *form* is the interface between the user and the database. It can be used just as you would use a form on a piece of paper, such as an order blank.

When you specify the type of frame you are creating, Vision creates a default form for that frame type. (See Frame Types (see page 65).) You can use that form as is or modify it.

Menus

Each operation available for a frame appears as a *menu item* on the frame's menu. These menu items let you view records and perform basic Ingres operations.

Vision generates these menu items in two ways:

- Vision automatically creates menu items that let users manipulate data and perform basic Ingres operations, such as saving changed records or returning to the top frame of an application. (The particular menu items that Vision generates depend on how you specify the visual query. For more information about Vision-generated menu items, see Vision Applications from a User's Perspective (see page 297).)
- When you create a new frame, you specify a menu item to call the new frame from an existing frame. For example the ChangeOrders menu item illustrated on the Browse frame later in this chapter was specified to call a frame where orders can be modified.

Frame Trees

The frames of a Vision application are arranged in groups called *trees*, because they resemble family trees. Within a tree, you refer to frames by their "family" relationship to each other.

A tree consists of:

- The *parent* frame; this is the highest-level frame from which you view a specific tree or tree segment
- The *child* frames; these are the frames that the parent frame calls

  Frames that have the same parent are called *peer* frames; peer frames also are said to be at the same level of the application.
- Any children of the child frames, and so on down the tree

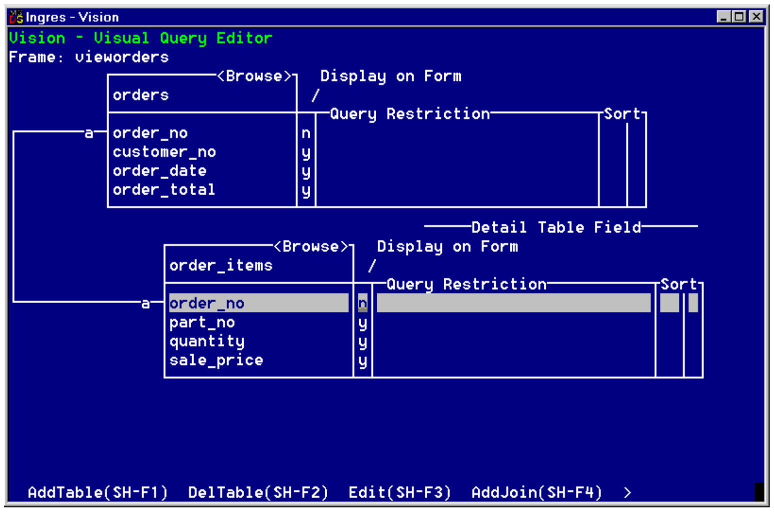
This "family tree" terminology is a convenient way to refer to the frames in an application. The procedures and discussions in this guide often use these terms. But remember that the terms parent and child refer to a relationship between frames, not to an absolute position in the application. As in a human family, a parent in one context can be a child in another.

Visual Queries

Vision applications query an Ingres database to retrieve, change, or input data. Vision generates the code for frames as a *visual query* that defines how the frame accesses the database.

Note: A visual query is also called the *frame definition*, and the process of specifying the query is referred to as *defining the frame*. These terms are used interchangeably in this guide.

Use the Vision Visual Query Editor to see the visual query. The following figure illustrates a visual query display window:



```
Vision - Visual Query Editor
Frame: vieworders
                          <Browse>  Display on Form
               orders              /
                                    Query Restriction            Sort
        a      order_no            n
               customer_no         y
               order_date          y
               order_total         y

                                                    Detail Table Field
                                <Browse>  Display on Form
                     order_items         /
                                          Query Restriction            Sort
               a     order_no            n
                     part_no             y
                     quantity            y
                     sale_price          y


  AddTable(SH-F1)  DelTable(SH-F2)  Edit(SH-F3)  AddJoin(SH-F4)  >
```

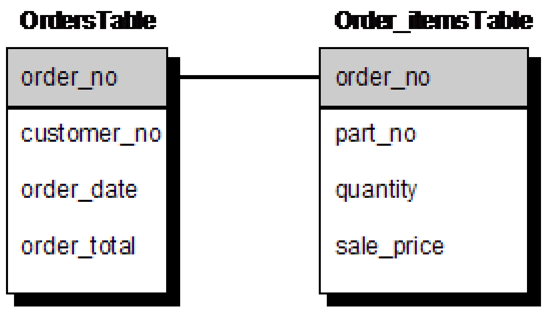
With the Visual Query Editor, you complete the following tasks:

- See the tables and columns used in a query.
- Indicate which columns to join for multi-table queries.
- Designate the columns to display as fields on the form.
- Assign values for specific fields or specify restrictions on the data retrieved by the query.
- Include Lookup tables in your queries to provide additional data to the user.

For information on how to use the Visual Query Editor to customize visual queries, see Defining Frames with Visual Queries (see page 117). For basic information about formulating Ingres queries, see the *Forms-based Querying and Reporting Tools User Guide.*

Joining Tables

Often the data you need for a query is located in more than one table. Ingres lets you *join* the tables that contain the information you need. This lets you combine data without having to include the same data in more than one table. The following figures demonstrate how two Ingres tables are joined. The Orders table contains information about each order; the Order_items table provides details about items to be ordered.



Assume you want to find how many of a certain part were ordered on a specific date. This information is not all available from one table. Your query needs to retrieve:

- All order numbers with that date from the Orders table
- The part number and quantities in the Order_items table with the order numbers that you retrieved from the Orders table

Join the tables on a column they have in common. This common column is referred to as the *join column*. In the example, the "order_no" (order number) column is the join column. You use the Visual Query Editor to specify the join columns. As illustrated in the Visual Query Editor above, Vision marks these columns so you can see how your tables are joined.

For more information about joining tables, see *Character-based Querying and Reporting Tools User Guide.* You may have used QBF already to specify join definitions (JoinDefs) for your tables. The concept of joining tables in Vision is identical, although the way you specify them is different.

Master and Detail Tables

In the previous example, each order appears once in the Orders table. The Order_items table, however, can contain many items that are part of a specific order. These tables are said to have a "one-to-many relationship."

In Ingres, tables that are related in this way are called *master* and *detail table*s*.* In the example, Orders is the master table and Order_items is the detail table.

When you use Vision to create a frame, you specify the master table. A detail table is optional.

Lookup Tables

A frame can have only one master table and one detail table, but you can include data from other tables by specifying these tables as *Lookup tables*.

Lookup tables are regular Ingres tables that generally contain information that is relatively fixed, such as part numbers or customer names and addresses.

When you include a Lookup table in a visual query, you specify how to display its data to the user. The part of the Lookup table that the user sees is called a *selection list*, because the application presents a list from which the user selects an item.

Selection lists are illustrated in Getting Started in Vision (see page 77). For a detailed explanation of how your applications can use selection lists, see Vision Applications from a User's Perspective (see page 297).

Frame Types

You develop a Vision application by creating frames that let the user perform specific operations. You can create frames of all types. The *frame type* controls the operations that the user can perform, for example, whether a user can add new records to a table or run a report.

A frame's type also determines how you must specify its database query:

- You specify some frames' queries with the Visual Query Editor. Vision generates the code and the form for these frames.
- Some frames use a query object—such as a table, report or graph—that you create with another Ingres tool. These tools specify the frames' queries, and, in some cases, specify the code that runs the frame.
- For some frames you must write the 4GL code and create the form.

Note: You can include procedures and call them just as you call frames. Any discussion of frame types in this guide refers to procedures as well.

The following table:

- Lists the frame types available in Vision
- Describes the function of each type
- Tells how you specify the query
- Tells how you specify the code and create the form

Examples of Vision-generated Frames (see page 68) shows sample visual queries and default forms for the frame types that Vision generates: Menu, Append, Browse, and Update.

The following table lists the frame and procedure types in Vision applications:

Type	What This Type Does	Where the Query Is Defined	How the Code and Form Are Created
Menu	Displays a list of operations to call other frames or to interact with the database.	Menu frames have no queries.	Vision generates the code and creates a default form that you can modify. (See the example in the next section.)
Append	Lets users insert new records into tables in the database.	You specify the query with Vision's Visual Query Editor.	Vision generates the code and creates a default form that you can modify. (See the example in the next section.)
Browse	Displays records retrieved from the master table and (optional) detail table; users cannot perform operations on the data retrieved. Browse frames can display a single record or all records that satisfy the	You specify the query with Vision's Visual Query Editor.	Vision generates the code and creates a default form that you can modify. (See the example in the next section.)

Type	What This Type Does	Where the Query Is Defined	How the Code and Form Are Created
	query. You can specify restrictions on the query to limit the data retrieved, or allow users to specify such restrictions, or both.		
Update	Performs all the functions described above for Browse frames; also lets users change the data retrieved. You also can let users perform these operations: ▪ Add new records ▪ Save a changed record as a new record, without affecting the original data ▪ Delete records You can specify any of these actions for the master and detail tables or the master table only.	You specify the query in Vision's Visual Query Editor.	Vision generates the code and creates a default form that you can modify. (See the example in the next section.)
report	Displays data in report format.	You specify the query for the report with RBF or the Report-Writer.	You can either: ▪ Use RBF to generate the code and a default report layout that you can modify ▪ Code and format the report with the Report-Writer Also, you can use VIFRED to create a frame on which the user can specify run-time parameters for the report.
QBF	Runs a query against a table or JoinDefinition.	You define the query with QBF.	QBF generates the code for the query. You can use a default QBF form or modify it with VIFRED.
User	Runs a user-defined frame.	You specify the query with 4GL	You must write the 4GL code and create the form with VIFRED.

Type	What This Type Does	Where the Query Is Defined	How the Code and Form Are Created
		statements.	
4GL Proc	Executes a 4GL procedure.	You write 4GL statements to define the operations that the procedure executes.	You must write the 4GL code. 4GL Procedures cannot refer to a form.
3GL Proc	Executes a procedure written in a 3GL (programming language) such as C or FORTRAN.	You write 3GL statements to define the operations that the procedure executes.	You must write the 3GL code. If the procedure uses a form, you must create it with VIFRED.
DB Proc	Runs a series of statements stored as a procedure in the database. The procedure must exist in the database before you can use it in a Vision application.	You write SQL statements to define the operations that the procedure must execute.	You must write the SQL statements. Database procedures cannot refer to a form.

Examples of Vision-generated Frames

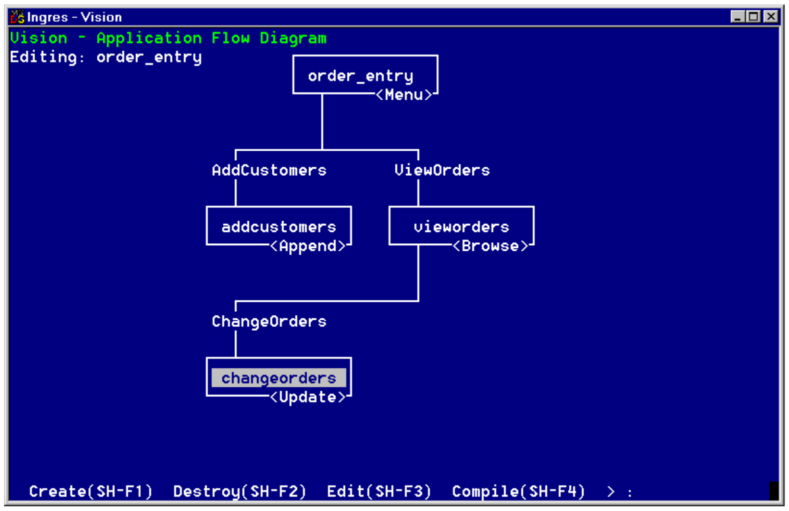
This section presents examples of an Application Flow Diagram and the four frame types that Vision generates: Menu, Append, Browse, and Update.

The examples represent a typical use for a Vision application—to enter and keep track of customer orders for a company. But the examples do not necessarily illustrate a complete order entry application. To perform additional functions, you could include many more frames, including ones that Vision does not generate, such as Report or User frames.

The Application Flow Diagram

The Application Flow Diagram is a visual tool for working with the frames of an application. It resembles an organization chart that shows how the frames of the application flow into each other so you can see how users move through the application.

The Application Flow Diagram is described in detail in Creating Frames (see page 87). It is included here to show how the four frames in the sample order entry application fit together:



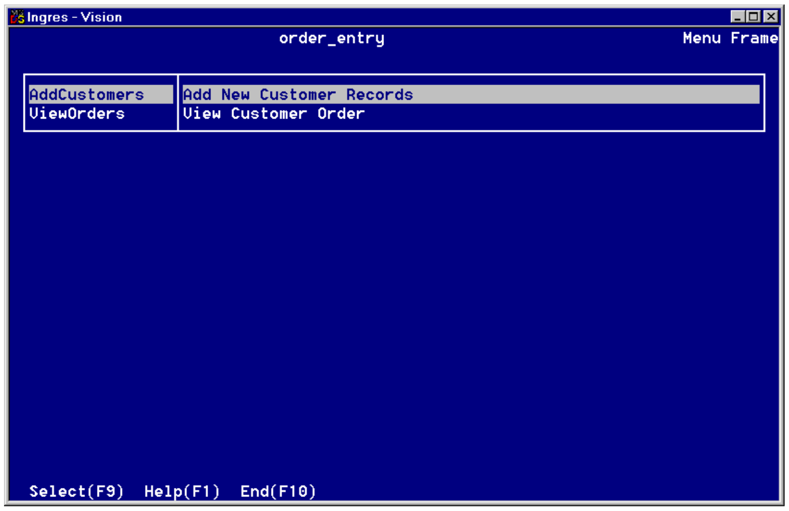
This Application Flow Diagram includes one frame of each type that Vision generates:

- A top Menu frame that calls the AddCustomers and ViewOrders frames
- AddCustomers: An Append frame to add new customer records
- ViewOrders: A Browse frame to view existing customer orders; this frame calls the ChangeOrders frame
- ChangeOrders: An Update frame to modify orders

Each of these frames is described in the following sections.

Menu Frame

When you create an application, Vision automatically generates a Menu frame as the top frame— starting point—of the application. This order entry menu frame illustrates the default form that Vision generates for the Menu frame:



```
                     order_entry                          Menu Frame

 AddCustomers   Add New Customer Records
 ViewOrders     View Customer Order

 Select(F9)  Help(F1)  End(F10)
```

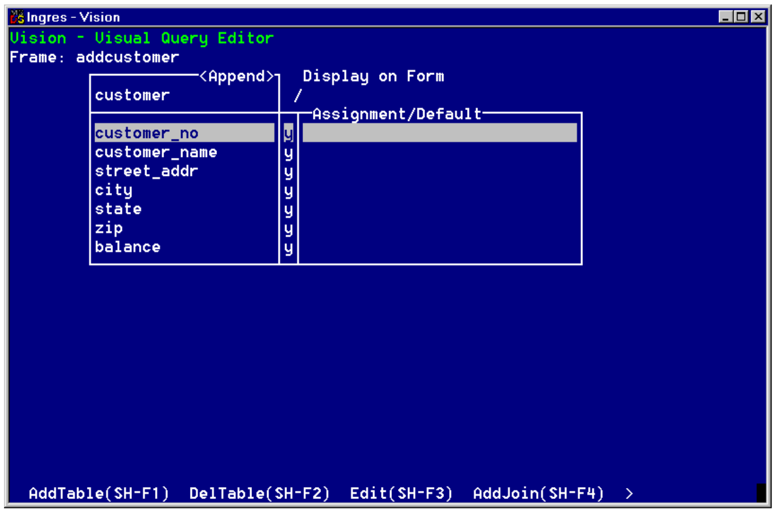
In the example, the Menu frame calls two other frames, AddCustomers and ViewOrders. The description of these frames comes from text that you specify when you create those frames.

By default, Vision places the menu items into a table field, but you can specify that a Menu frame be displayed as a *single-line menu*; that is, with the menu items displayed in a line along the bottom of the frame. For more information about specifying Menu frames, see Creating Frames (see page 87).

A Menu frame does not have a visual query.

Append Frame

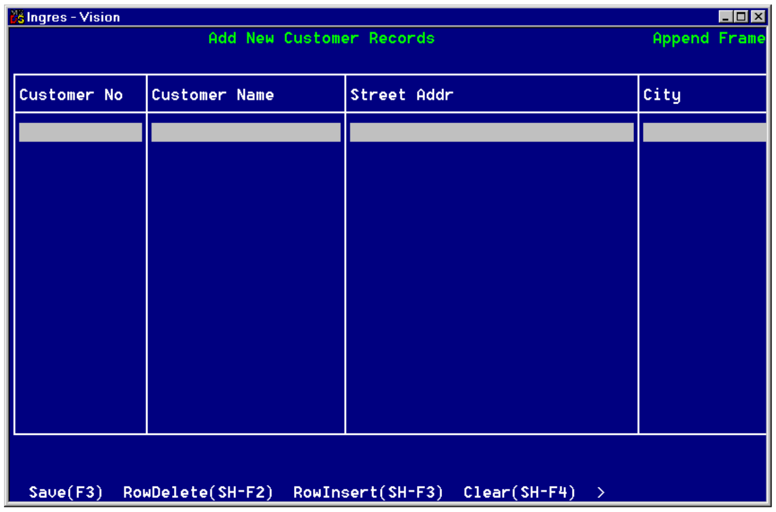
AddCustomers is an Append frame that lets users add new records to the Customer table. The following figure shows the visual query for this frame:



```
Vision - Visual Query Editor
Frame: addcustomer
                       ┌<Append>┐ Display on Form
 customer              │  /
                       │ ┌Assignment/Default─────────────┐
 customer_no          │y│                                │
 customer_name        │y│                                │
 street_addr          │y│                                │
 city                 │y│                                │
 state                │y│                                │
 zip                  │y│                                │
 balance              │y│                                │

 AddTable(SH-F1)  DelTable(SH-F2)  Edit(SH-F3)  AddJoin(SH-F4)  >
```

This frame only uses a master table.

The following figure shows the form that Vision generates for this Append frame:

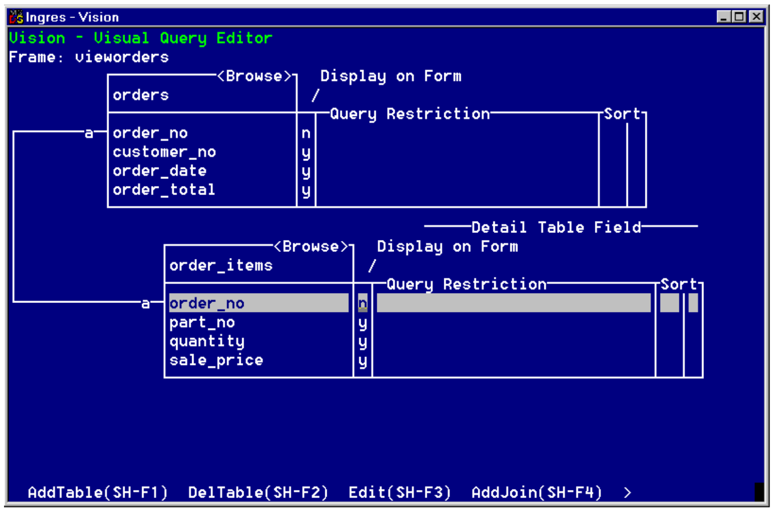


```
                    Add New Customer Records                  Append Frame

 Customer No │ Customer Name      │ Street Addr                │ City

 Save(F3)  RowDelete(SH-F2)  RowInsert(SH-F3)  Clear(SH-F4)  >
```

The form contains a table field with a column for each column in the Customer table. The frame also contains the necessary menu items to let users add records and perform other standard operations, like saving, deleting, or clearing.

Browse Frame

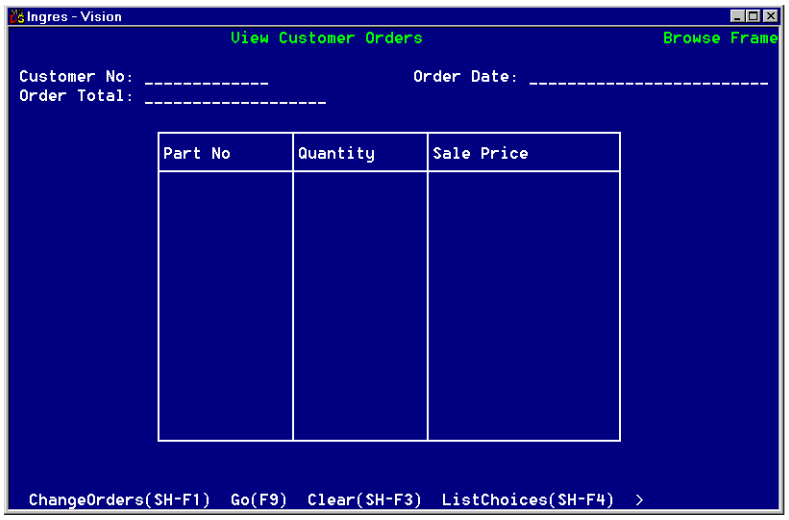
ViewOrders is a Browse frame that lets users display existing customer orders. The following figure shows the visual query for this frame:



```
Ingres - Vision
Vision - Visual Query Editor
Frame: vieworders
                  <Browse>  Display on Form
     orders                /
                            Query Restriction             Sort
  a  order_no        n
     customer_no     y
     order_date      y
     order_total     y
                             Detail Table Field
                  <Browse>  Display on Form
     order_items           /
                            Query Restriction             Sort
  a  order_no        a
     part_no         y
     quantity        y
     sale_price      y

 AddTable(SH-F1)  DelTable(SH-F2)  Edit(SH-F3)  AddJoin(SH-F4)  >
```

The frame uses the Orders table as the master table and the Order_items table as the detail table.

The following figure shows the form that Vision generates for this frame:



```
Ingres - Vision
                   View Customer Orders                     Browse Frame

 Customer No: _____________             Order Date: _________________________
 Order Total: ___________________

                Part No       Quantity      Sale Price

 ChangeOrders(SH-F1)  Go(F9)  Clear(SH-F3)  ListChoices(SH-F4)  >
```

The form contains:

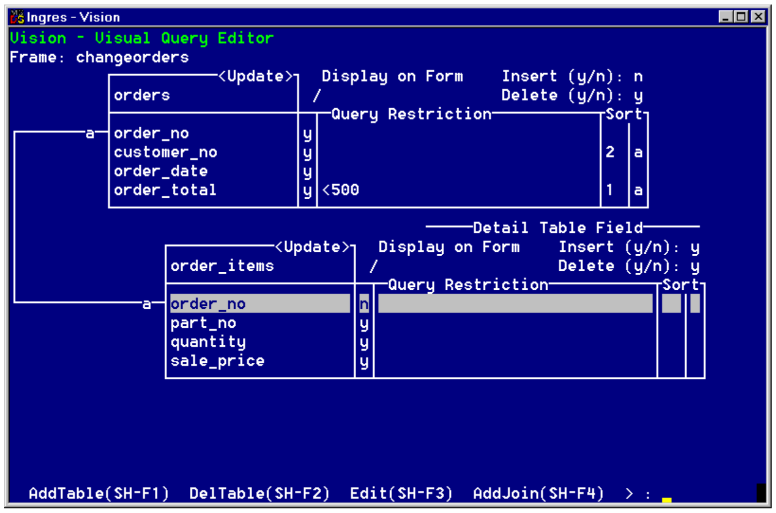
- A simple field for each column in the Orders table
- A table field to display all the detail records in the Order_items table

The Browse frame also contains the necessary menu items to let users retrieve records and perform other standard operations. Remember that users can only display records on a Browse frame. Therefore, in this example, the menu line contains a ChangeOrders menu item to call the Update frame, on which users can change an order.

Update Frame

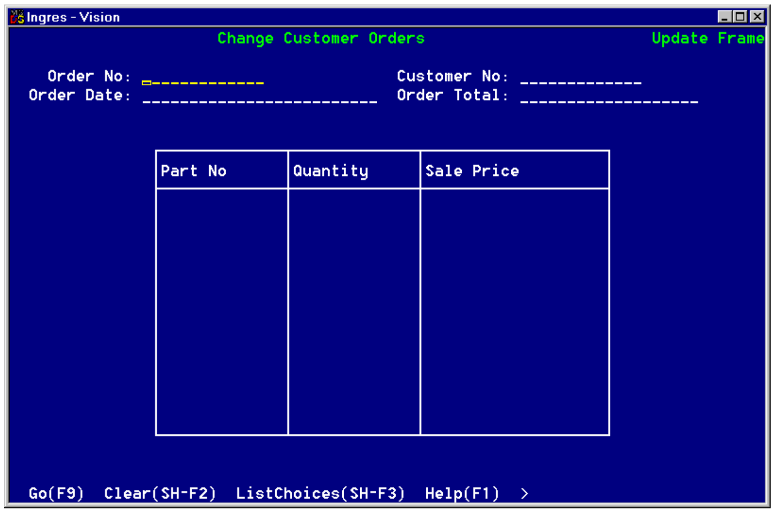
The final frame in the example is an Update frame called ChangeOrders. This frame lets users modify information about a customer order, such as changing the quantity of an item.

The following figure shows the visual query for the Update frame. It looks similar to the visual query for the Browse frame, using Orders as the master table and Order_items as the detail table. However, the visual query for an Update frame also lets you specify whether users can add new records or delete records in addition to changing existing data.



```
Ingres - Vision
Vision - Visual Query Editor
Frame: changeorders
                      <Update>  Display on Form    Insert (y/n): n
          orders                 /                 Delete (y/n): y
                                   Query Restriction           Sort
        a  order_no           y                                
           customer_no        y                           2  a
           order_date         y
           order_total        y <500                      1  a

                                            Detail Table Field
                            <Update>  Display on Form    Insert (y/n): y
                order_items            /                 Delete (y/n): y
                                         Query Restriction           Sort
              a order_no            n
                part_no             y
                quantity            y
                sale_price          y

  AddTable(SH-F1)  DelTable(SH-F2)  Edit(SH-F3)  AddJoin(SH-F4)  > :
```

The following figure illustrates the form that Vision generates for the Update frame:



```
Ingres - Vision
                    Change Customer Orders                          Update Frame

  Order No: _____________                         Customer No: _____________
Order Date: _________________________            Order Total: ___________________

                +-------------+-------------+--------------------+
                |Part No      |Quantity     |Sale Price          |
                +-------------+-------------+--------------------+

  Go(F9)  Clear(SH-F2)  ListChoices(SH-F3)  Help(F1)  >
```

This form looks similar to the form for the Browse frame, because both frames have similar visual queries. But Vision generates different menu items to let users perform the additional operations allowed on an Update frame

Note: You don't see these menu items on the sample form, because they are not visible when a user calls the frame; they appear after the user selects a record to update.

Ingres Data Types

When you create a Vision frame based on an Ingres database table, the visual query and form contain *fields* that correspond to *columns* in the database table. These columns have *data types* that control the type of information that can be contained in the column and how Ingres handles that information. The names and data types of the fields are taken from the table's definition.

The following table lists the data types that Vision uses and the situations where they can be used:

Usage	**Char varchar**	**Small int integer**	**decimal**	**none**	**string**	**money date**	**float**	**record type array**
User frame return type	X	X	X	X	X	X	X	
4GL proc return type	X	X	X	X	X	X	X	

Usage	**Char varchar**	**Small int integer**	**decimal**	**none**	**string**	**money date**	**float**	**record type array**
3GL proc return type	X	X		X	X		X	
Database proc return type		X		X				
Local proc return type	X	X	X	X		X	X	
Local variables	X	X	X			X	X	X
Global variables	X	X	X			X	X	X
Record type attributes	X	X	X		X	X	X	X
Global constants	X	X	X				X	

Vision supports the standard Ingres data types, with the exception of long varchar, long bit, byte, byte varying, and long byte. If a Vision frame is based on a table that has a column of one of those types, the column is not displayed or included in the visual query. The remaining columns in the table are displayed.

For more information, see Data Types (see page 46), Variables (see page 46), Constants (see page 47), and Record Types (see page 48).

Chapter 4: Getting Started in Vision

This section contains the following topics:

This chapter covers how to get started with Vision.

Start Vision

Vision can be started directly from the operating system prompt or by using the Ingres menu option.

To start Vision from the operating system prompt

1. At the operating system prompt, type:

   `vision [`*nodename*`::]`*dbname*

   where *dbname* is the name of the database you are using. If you are using a remote host, *nodename* is the name of the remote node.

2. Press Return.

   Vision displays the Applications Catalog window. This is illustrated in Overview of Tools and Languages (see page 35).

   For the full syntax of the vision command, see Using Vision or ABF in a Multi-Developer Environment (see page 1381).

To start Vision through the Ingres Menu

1. At the operating system prompt, type:

   ```
   ingmenu [nodename::]dbname
   ```

   where *dbname* is the name of the database you are using. If you are using a remote host, *nodename* is the name of the remote node.

2. Press Return.

   Ingres Menu is displayed.

3. Select Applications.

   Vision displays the Applications Catalog window. This is illustrated in Overview of Tools and Languages (see page 35).

   When you exit Vision, you return to the Ingres Menu.

Call a Specific Application

To work on an existing application in your database, start Vision to display the application flow diagram for that application.

To go directly to an application, include the application name in the Vision command by entering it at the operating system prompt:

```
vision [nodename::]dbname [applicationname [framename]]
```

You also can include the name of a specific frame after the application name. If you do, Vision runs the application from the frame that you specify, rather than letting you edit it. For more information, see Vision Applications from a User's Perspective (see page 297).

If you are working on the same applications as other developers, you must access Vision in specific ways to be able to share your applications. For more information, see Using Vision or ABF in a Multi-Developer Environment (see page 1381). Other command-line parameters used to access Vision in various ways are also listed in this appendix.

Stop Vision

To end a Vision session, select Quit from the menu. Vision returns you to the operating system prompt.

If the window does not have a Quit operation, select End. Vision returns you to the next higher-level window. Continue selecting End until a window displays the Quit option.

See How to Choose a Menu Operation (see page 80) for information on selecting menu operations.

Save any changes you have made. In most windows, selecting Quit or End automatically saves your changes, however, on some windows, you must use the Save operation to save your changes.

Vision provides a warning if you try to leave a window without saving your changes.

Moving Around in Vision

You can navigate in Vision by using keys, choosing operations from menus, and by using pop-up windows and selection lists as described in the following sections.

How to Use Keys

Use the following keys to move around the window:

- The Tab key to move forward between fields in the window
- The Previousfield key to move backward between fields
- The up and down arrow keys to move between rows in a table

Use the Keys option of the Help operation from any Vision window. This option displays a list of the keys to use to perform the operations available for the window. See How to Use Vision's Online Help (see page 84) for more information about the Help Keys option.

How to Choose a Menu Operation

In each window, Vision presents a menu of operations. Choose an operation from the menu in one of these ways:

- Use the Menu key to move the cursor to the menu line, then type the first letter of the operation you want.

  If two or more operations begin with the same letter, you must type at least the first distinct letters of the operation. For example, if a menu displays both Create and Cancel operations, type cr for create or ca for cancel.

- If a menu item contains a number or symbol in parentheses after the operation name, press the corresponding number or symbol on the keypad to execute the operation.

  For example, if the menu contains the menu item Save(0), press the 0 key on the keypad to execute the Save operation.

  Note: For this function to work, your keypad keys must be mapped to the corresponding operations. See *Character-based Querying and Reporting Tools User Guide* for more information on PC or terminal mapping.

Display Hidden Menu Items

When all the available menu items cannot fit on the window, Vision displays the ">" symbol at the end of the menu line to indicate that additional choices are available.

To see the hidden menu items

1. Press the Menu key.

   The cursor moves to the menu line.

2. Press the Menu key again.

   Additional menu items appear. Also, the "<" symbol appears to the left of the menu line. This indicates that the original menu items have scrolled out of the window.

   To make these menu items reappear, cycle through the menu again by pressing the Menu key.

3. If the ">" symbol still is visible, there are more menu items that have not appeared in the window yet. Press the Menu key again to see these additional menu items.

Some operations display another menu, or *submenu*, containing more available operations. Select a menu item from a submenu in the same way as from a regular menu.

After completing an operation from the submenu, Vision returns you to the original menu.

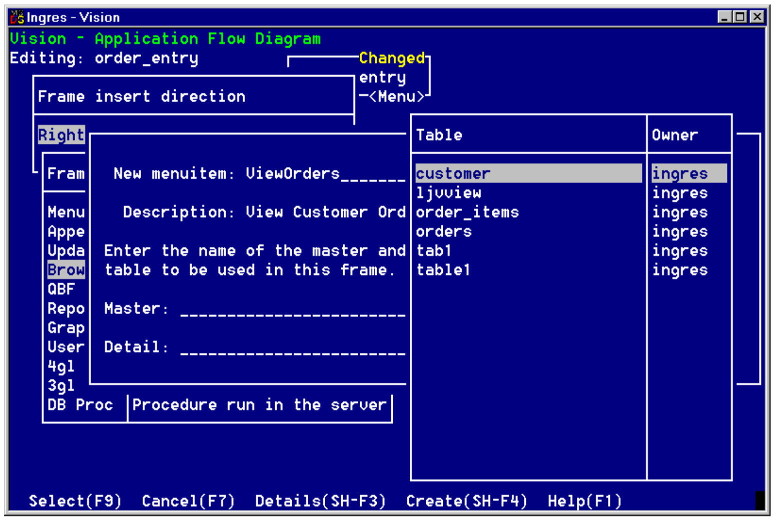
How to Use Pop-Up Windows and Selection Lists

Vision displays some information on *pop-up windows* that cover only part of the window underneath them. When you finish working with a pop-up window, Vision returns you to the original window to continue your operation.

A pop-up window can:

- Display information or warning messages
- Prompt you for a response
- Display a selection list from which you can choose an item

The following figure shows a selection list for choosing a table during the frame creation process:



```
Ingres - Vision
Vision - Application Flow Diagram
Editing: order_entry          ┌──────Changed┐
                                       entry
  Frame insert direction              ─<Menu>┘

  Right                          Table              Owner
  Fram   New menuitem: ViewOrders_______  customer   ingres
                                          ljvview    ingres
  Menu    Description: View Customer Ord  order_items ingres
  Appe                                    orders     ingres
  Upda  Enter the name of the master and  tab1       ingres
  Brow  table to be used in this frame.   table1     ingres
  QBF
  Repo  Master: ________________________
  Grap
  User  Detail: ________________________
  4gl
  3gl
  DB Proc  Procedure run in the server

 Select(F9)  Cancel(F7)  Details(SH-F3)  Create(SH-F4)  Help(F1)
```

When choosing an item from a selection list, Vision automatically enters the item you select into the appropriate field.

To use a selection list

1. Position the cursor on the field for which you want to display the selection list.
2. Select ListChoices from the menu.

   Vision displays the selection list (illustrated in the preceding figure).
3. Move the cursor to the item you want in one of the following ways:
   - Use the up or down arrow.
   - Enter the first letter of the item you want. The cursor jumps to the next item in the list that begins with that letter.

     Note: To search the entire list using this function, you must scroll to the end of the list once; otherwise, only a partial list consisting of the buffered rows is searched.
   - Use the Find operation to move the cursor to a specific text string.

     In Vision, the Find operation does not appear on the menu. You must use the key combination mapped to the Find operation. To see the key mapping, select Help from the menu, then select Keys from the submenu.
4. Choose Select from the menu or press Return.

   When you select from a list of Ingres data types, you are prompted for more information as follows:
   - If you select a character data type, Vision prompts you for the length of the data type.
   - If you select the decimal type, Vision prompts you for the precision and scale.

   In the above situations:

   a. Enter the requested information (without parentheses) on the pop-up.

   b. Choose OK to enter the data type in the field.

   Vision enters the item you select into the field.

   To return to the previous window without making a selection, choose Cancel (or End on some windows).

The selection lists that you use in Vision work the same way as the selection lists that you provide in your applications when you include Lookup tables in your visual queries, as described in Defining Frames without Visual Queries (see page 159).

How to Handle Error Messages

If you make an error while performing an operation in Vision, an error message is displayed at the bottom of the window. The message gives you basic information about errors in the operation that you just performed.

After reading the message, press the Return key to clear it from the window, correct the error, and continue developing your application.

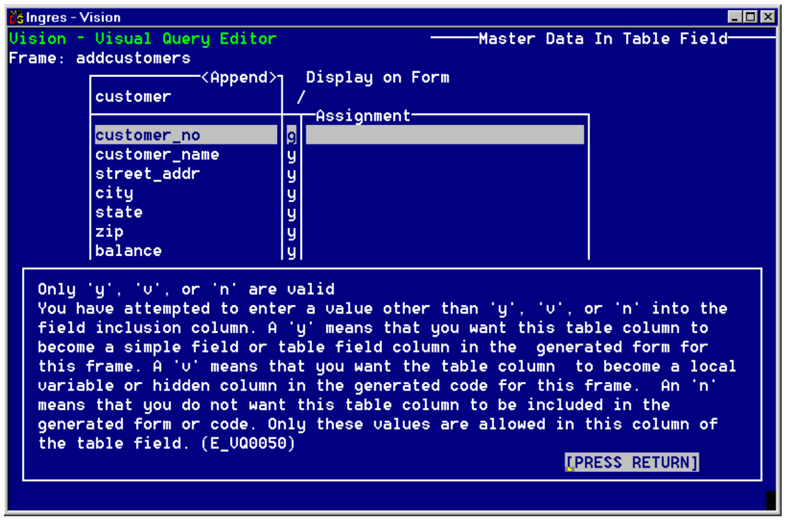
Display Multi-line Error Messages

Vision displays error messages on one line. If the error message is too long to fit on one line, you can display a pop-up window with more information about the error.

To display a multi-line error message in a pop-up window

1. Read the initial error message.
2. Select More by pressing the h key or the function key named in the window.

   The pop-up window now displays an explanation of the error and how to correct it.



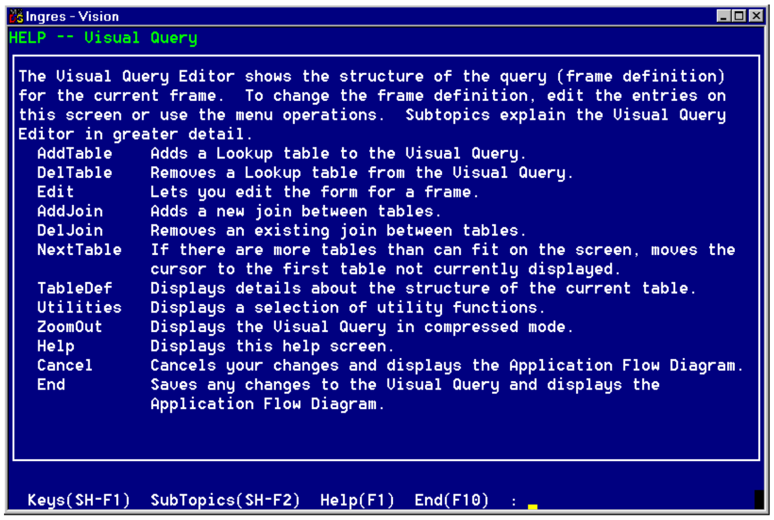
```
Ingres - Vision
Vision - Visual Query Editor              ——Master Data In Table Field——
Frame: addcustomers
         ┌──────────<Append>┐  Display on Form
         │customer          │ /
         └──────────────────┘┌Assignment──────────────────┐
         │customer_no       │g│                           │
         │customer_name     │y│                           │
         │street_addr       │y│                           │
         │city              │y│                           │
         │state             │y│                           │
         │zip               │y│                           │
         │balance           │y│                           │

  Only 'y', 'v', or 'n' are valid
  You have attempted to enter a value other than 'y', 'v', or 'n' into the
  field inclusion column. A 'y' means that you want this table column to
  become a simple field or table field column in the  generated form for
  this frame. A 'v' means that you want the table column  to become a local
  variable or hidden column in the generated code for this frame.  An 'n'
  means that you do not want this table column to be included in the
  generated form or code. Only these values are allowed in this column of
  the table field. (E_VQ0050)
                                                        [PRESS RETURN]
```

   After reading the explanation, press Return to return to the original window.

How to Use Vision's Online Help

Each Vision window includes a Help operation on its menu. Use the help facility at any time to read more information to perform a function used in developing a Vision application.

The following figure shows a Vision help window:



```
Ingres - Vision
HELP -- Visual Query

The Visual Query Editor shows the structure of the query (frame definition)
for the current frame.  To change the frame definition, edit the entries on
this screen or use the menu operations.  Subtopics explain the Visual Query
Editor in greater detail.
  AddTable     Adds a Lookup table to the Visual Query.
  DelTable     Removes a Lookup table from the Visual Query.
  Edit         Lets you edit the form for a frame.
  AddJoin      Adds a new join between tables.
  DelJoin      Removes an existing join between tables.
  NextTable    If there are more tables than can fit on the screen, moves the
               cursor to the first table not currently displayed.
  TableDef     Displays details about the structure of the current table.
  Utilities    Displays a selection of utility functions.
  ZoomOut      Displays the Visual Query in compressed mode.
  Help         Displays this help screen.
  Cancel       Cancels your changes and displays the Application Flow Diagram.
  End          Saves any changes to the Visual Query and displays the
               Application Flow Diagram.

  Keys(SH-F1)  SubTopics(SH-F2)  Help(F1)  End(F10)  :
```

After viewing the main help window about a function, obtain additional information in these areas:

Keys

Explains what keys you can use on the current window

SubTopics

Displays information related to the function you are performing

Display Help

You can display help for a Vision window.

To get help about a Vision window

1. Select Help from the menu on the window.

   Vision displays a window of information about the operation you are performing.

2. View additional information as follows:

   - To view information on related subtopics, see the following procedure.
   - To view general information about the help facility, select Help from the menu.
   - To find out what keys you can use in the window on which you were working, select Keys from the menu.

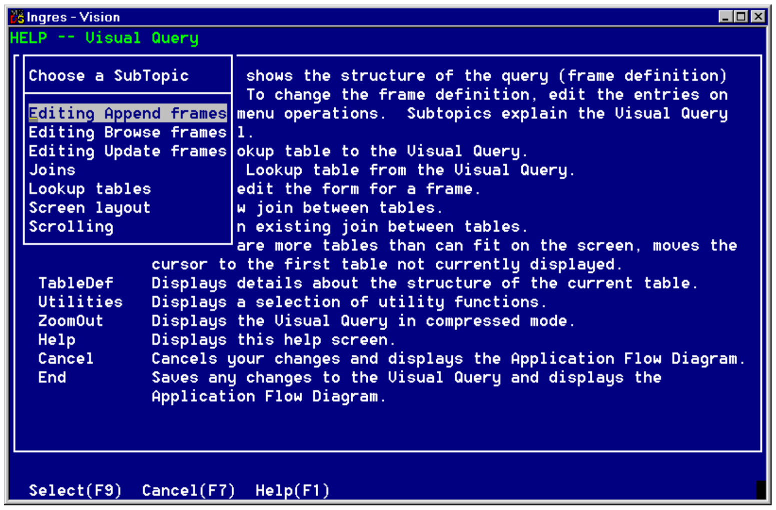
   After viewing any of these additional help windows, select End to return to the main help window.

3. To return to the original window, in which you were working, select End from the help window menu.

To view information on related subtopics

1. Select SubTopics from the help window menu.

   Vision displays a list of the related subtopics:



```
Ingres - Vision
HELP -- Visual Query

 Choose a SubTopic     shows the structure of the query (frame definition)
                        To change the frame definition, edit the entries on
 Editing Append frames menu operations.  Subtopics explain the Visual Query
 Editing Browse frames l.
 Editing Update frames okup table to the Visual Query.
 Joins                  Lookup table from the Visual Query.
 Lookup tables         edit the form for a frame.
 Screen layout         w join between tables.
 Scrolling             n existing join between tables.
                       are more tables than can fit on the screen, moves the
              cursor to the first table not currently displayed.
  TableDef    Displays details about the structure of the current table.
  Utilities   Displays a selection of utility functions.
  ZoomOut     Displays the Visual Query in compressed mode.
  Help        Displays this help screen.
  Cancel      Cancels your changes and displays the Application Flow Diagram.
  End         Saves any changes to the Visual Query and displays the
              Application Flow Diagram.

  Select(F9)  Cancel(F7)  Help(F1)
```

2. Position the cursor on a topic and choose Select from the menu.

   Vision displays the help window for the topic.

3. Exit the subtopic help window in either of these ways:

   - Select PrevTopic from the menu to return to the previous help window.
   - Select End to return to the window in which you were working.

Chapter 5: Creating Frames

This section contains the following topics:

This chapter describes using the Application Flow Diagram Editor to create new frames and work with existing frames after you create the application itself.

How to Call the Application Flow Diagram Editor

Access the Application Flow Diagram Editor by:

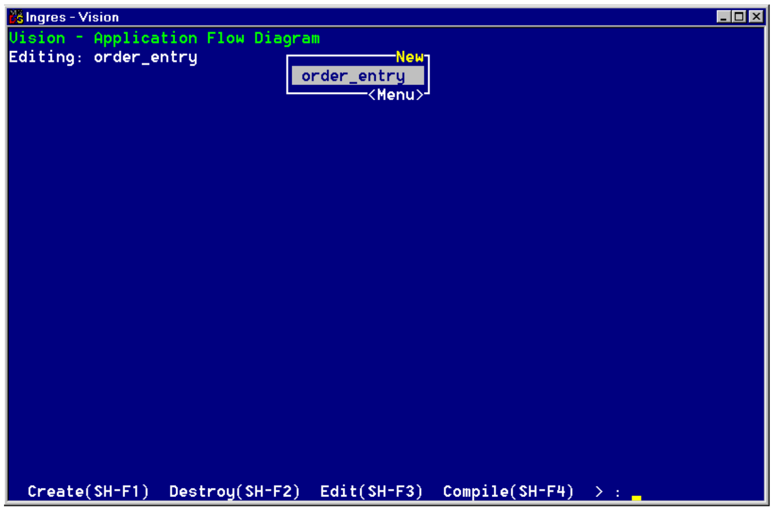
- Creating a new application
- Editing an existing application

Each of these methods is described in the sections that follow.

Create a New Application

Create a new application in the Applications Catalog, as described in Overview of Tools and Languages (see page 35). When the new application is created, Vision opens the Application Flow Diagram Editor window.

The following figure shows the application flow diagram for a new application.



```
Vision - Application Flow Diagram
Editing: order_entry
                                        New
                                  order_entry
                                        <Menu>

  Create(SH-F1)  Destroy(SH-F2)  Edit(SH-F3)  Compile(SH-F4)  > :
```

At this point in the development process, the application contains only a Menu frame, which Vision creates automatically as the top frame of the application; it uses the name of the application as the name of the frame. This top frame is the starting point for your application.

Note: Vision applications reside in the same database as the data they access.

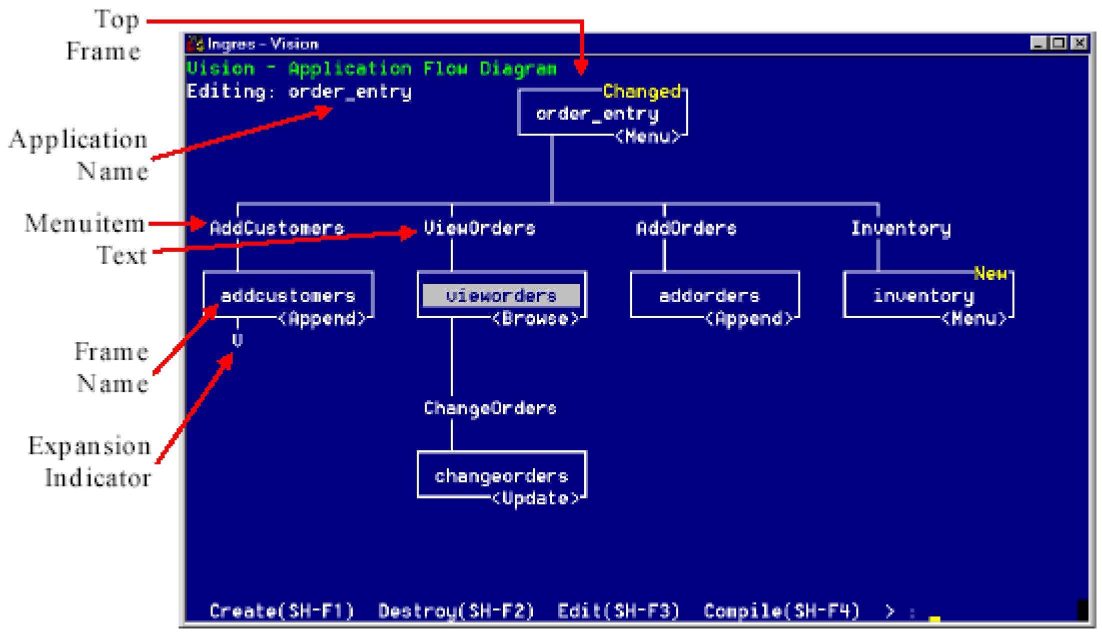
Edit an Existing Application

Use the application flow diagram to display the structure of an existing application, create new frames for the application, and work with the existing ones.

To call the Application Flow Diagram Editor for an existing application

1. Position the cursor on the name of the application on the Applications Catalog window.
2. Select Edit from the menu.

The following figure shows the application flow diagram for an existing application:



This window is described in detail in the next section.

How You Can Use the Application Flow Diagram

The Application Flow Diagram Editor displays the application flow diagram, which is a workspace in which to build your application. The application flow diagram is the workspace in which you create the frames for your application. It provides a view of the structure of the application as it appears to the user and allows you to accomplish the following:

- View how the frames are connected
- Create and destroy frames
- Remove and insert frames
- Specify and change menu item text
- Edit various components of the frames

The application flow diagram immediately displays any changes to the structure of your application.

Details in the Application Flow Diagram

Vision displays the application flow diagram and displays the following information about the application:

- The name of the application is displayed in the upper left of the window.
- Each box represents a frame of the application.
- The name of the frame with which you are working is highlighted. This frame is called the current frame.
- The first 14 characters of the frame's name are displayed in its box. If you are using Microsoft Windows, the top frame displays 8 characters. To see an entire frame name that is longer than 14 characters, use the MoreInfo operation of the Application Flow Diagram Editor menu.
- The frame's type is displayed inside the brackets (< >) at the bottom right of the frame's box.
- The menu item text that appears above a box represents the menu choice that calls this frame from the parent frame.
- Any status indicator—New, Changed, Custom or Error—appears in the upper right of the box. (See Frame Status (see page 94).)
- The application flow diagram displays the child frames for the current frame only. The "v" expansion indicator shows you where the diagram can be expanded to show the children of the peer frames of the current frame.
- The expansion indicator ("<" for left or ">" for right) at the end of a row of peer frames indicates that there are additional peer frames that do not appear in the window.

  A standard terminal screen can display three levels down and four frames across at a time. (If you have a larger screen, you can display more of the application flow diagram.) See Ways to Expand the Application Flow Diagram (see page 91) for instructions on displaying additional frames.

In the preceding figure, the application flow diagram shows the child of the "vieworders" frame. The "v" below its peer frame "addcustomers" indicates that this frame also has children. To view the children of "addcustomers," you must make this frame the current frame, as described below.

Also, the expansion indicator (">") above and to the right of the "Inventory" menu item indicates that there are additional peer frames at this level. Scroll horizontally to the right to view these frames.

Use the operations on the Application Flow Diagram Editor window menu to create and manipulate frames as described in this chapter. For a full list of the Application Flow Diagram Editor menu operations, see Menu Operations (see page 1375).

Select the Current Frame

Before you can perform any operations on a frame in the application flow diagram, you must select that frame as the *current frame*. Vision highlights the frame you select.

You can select the current frame in either of these ways:

- Move to the current frame with the up, down, left, or right arrow keys.
- Use the Find operation to specify all or part of a frame name. The first frame name that contains the text you entered becomes the current frame.

  The Find operation does not appear on the menu. You must use the key combination mapped to the Find operation. To see the key mapping for your Vision installation, select Help from the menu, then select Keys from the submenu.

Ways to Expand the Application Flow Diagram

If the entire application flow diagram does not fit in the window, Vision displays the current frame and:

- The parent of the current frame (except when the top frame is the current frame)
- Up to four of the children of the current frame
- Up to three of the peers of the current frame

There are three ways to display additional frames:

- Expanding a peer frame
- Horizontal scrolling
- Vertical scrolling

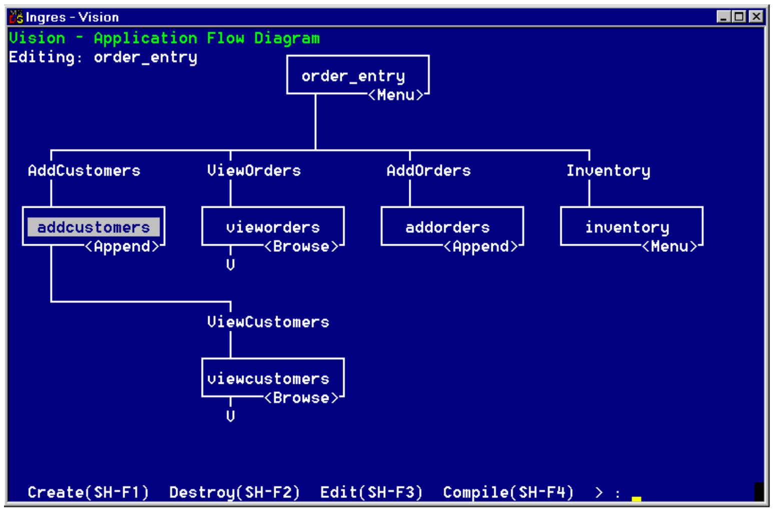
Each of these methods is described in the following sections.

How You Can Expand a Peer Frame

To view the children of a peer of the current frame, use the right or left arrow key so that the peer becomes the current frame. The application flow diagram display changes so that:

- The frame you selected is highlighted
- Up to four children of this frame are displayed
- The children of the previous current frame disappear
- The previous current frame is marked with the "v" expansion indicator

The following figure shows how the application flow diagram in the previous figure is redisplayed when you change the current frame from ViewOrders to its peer, AddCustomers:



```
Vision - Application Flow Diagram
Editing: order_entry
                                    order_entry
                                        <Menu>

  AddCustomers       ViewOrders        AddOrders         Inventory

   addcustomers       vieworders        addorders         inventory
      <Append>          <Browse>          <Append>           <Menu>
                     v

                     ViewCustomers

                     viewcustomers
                        <Browse>
                     v

  Create(SH-F1)  Destroy(SH-F2)  Edit(SH-F3)  Compile(SH-F4)  > :
```

How You Can Scroll Horizontally

Frames can be scrolled off the window as you move to the left or right among peer frames. When this happens, the left or right expansion indicator is displayed to indicate where frames are not visible.

To see the invisible peers of the current frame, use the right or left arrow keys to scroll horizontally until the frame you want as the current frame appears. Notice that as you scroll horizontally among peer frames, only the children of the *current* frame remain visible.

How You Can Scroll Vertically

Use the up or down arrow keys to select the level of the frame you want. If the application flow diagram contains more than three levels of frames, frames can disappear when you change the current frame to a different level:

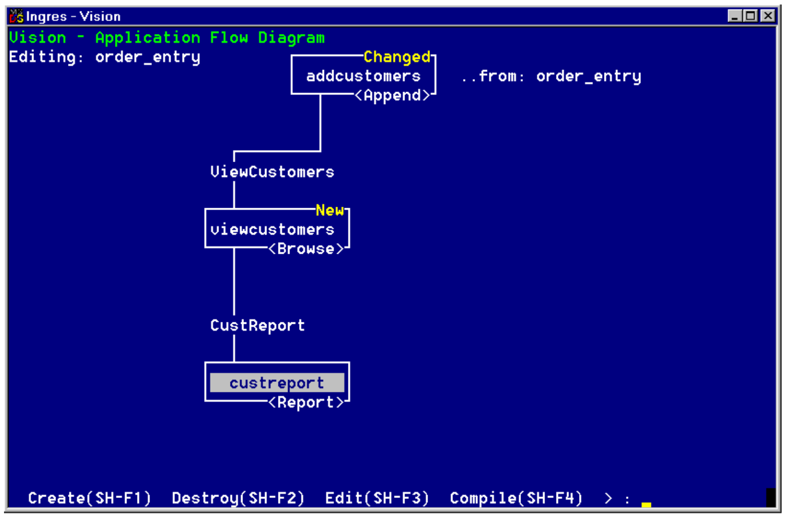
- When you move to a higher level, the bottom level of the current display can disappear.

  The "v" expansion indicator shows you where frames have been scrolled off the bottom of the window.

- When you move to a lower level, the top level of the display can disappear.

  If this happens, the message "from *frame_name*" appears in the upper right of the application flow diagram (where *frame_name* represents the name of the parent frame that disappeared).

  This figure shows what happens when you scroll down from "viewcustomers" to "custreport" as the current frame:



```
Vision - Application Flow Diagram
Editing: order_entry
                                          Changed
                                   addcustomers      ..from: order_entry
                                          <Append>

                         ViewCustomers
                                     New
                         viewcustomers
                                  <Browse>

                         CustReport

                           custreport
                                  <Report>

  Create(SH-F1)  Destroy(SH-F2)  Edit(SH-F3)  Compile(SH-F4)  > :
```

If you move to the top frame or one of the bottom-level frames, the display does not change in these ways, because there are no additional levels to scroll into the application flow diagram.

Frame Status

Vision sometimes displays an indicator in the upper right corner of a frame's box to show its current status. As an example, in the preceding figure, the Changed frame status indicator appears for the frame "addcustomers."

The following table lists the indicators that are displayed.

Status Indicator	**This Indicator Appears**
New	Until you first compile or test a frame
Changed	Whenever you: ▪ Add or delete child frames ▪ Change the menu item text ▪ Change a frame's visual query ▪ Add or change escape code or parameters The Changed indicator means that Vision is going to recompile the frame before running it. This indicator is not displayed on a frame that you move to a new position in the application flow diagram, but does appear on the old parent and new parent of a frame that you move (because you have deleted and added a child frame, respectively, to the old and new parent frames).
Custom	If you have edited the Vision-generated code for the frame. The Custom indicator remains until you regenerate the code.
Error	If errors are found in the 4GL code when you compile a frame. The Error indicator remains until you correct the errors and recompile the frame. For more information about correcting errors in 4GL code, see Completing a Vision Application (see page 259).

Create New Frames

You use the Application Flow Diagram Editor to create new frames for an application. The following list summarizes the procedure used to create frames. For detailed instructions and illustrations, see Create a New Frame (see page 98).

To create a new frame

1. Specify the frame's position in the application flow diagram, relative to the current frame.

   The positions of the frames determine the order in which the user can call them. You can create frames as peers of the current frame or, in some cases, as children of the current frame. Only Menu, Append, Browse, and Update frames can have children that are displayed in the application flow diagram.

2. Indicate the frame type.

   The frame's type determines the operations that the user can perform on the frame. For a description of each frame type, see Frame Types (see page 65).

3. Specify the menu item text that calls the frame.

   This text appears on the menu of the parent frame that calls the frame you are creating.

4. Enter a description of the frame.

   This description appears on any Menu frames that call the new frame. It also appears in the code and on the form that Vision generates and in various Vision windows.

5. For Append, Browse, and Update frames, designate the database tables to use for the frame's visual query.

   Vision uses the tables when it creates the default frame definition. Read Specifying Tables (see page 96) before you create any Append, Browse, or Update frames.

6. For frames with types other than Menu, Append, Browse, and Update, define the frame as described in Defining Frames without Visual Queries (see page 159).

Specifying Tables

When you create an Append, Browse, or Update frame, Vision identifies the tables in the database used for the frame definition. The tables can exist in the database before you create the frame, or you can create the tables when you create the frame.

Note: You can create a frame that refers to a *synonym* or a *view*, instead of a table name. A synonym is an alternate name for a table. A view is a virtual table. All discussion about tables in this manual can apply to synonyms and views, except that synonyms and views cannot be created with the Ingres Tables Utility or within Vision as described in Create a New Frame (see page 98). You use query language statements to create synonyms and views. For more information, see your query language manual.

The procedures in Create a New Frame (see page 98) describe the ways in which you can specify Master and Detail tables when you create a Vision frame. Whichever method you use, the following rules apply:

- You must designate one table as the Master table.
- The Detail table is optional.
- You cannot use more than one Master table and one Detail table.
- If you designate that the Master table appear on the form as a table field, then you cannot include a Detail table for the frame.

  If you use a table field for the Master table on a Browse or Update frame, you cannot enable the Next Master Menu item frame behavior. For a discussion of these frame behaviors, see Defining Frames with Visual Queries (see page 117).
- You cannot change your Master and Detail table designations after you create the frame. If you must change the table names or add a Detail table, you must destroy the frame (as described below) and then recreate it.
- If you change the underlying structure of a table you have specified for a frame—for example, by adding columns or changing data types—you should use the table reconciliation utility described in Reconciling Tables and Frame Definitions (see page 251). This utility lets you make sure that the frame definition corresponds to the new table definition.

- If the frame you are creating uses the same Master and Detail tables as an existing frame, you can copy the frame definition from the existing frame. To do this, you can use the Duplicate or Import operation.

  Use the Duplicate operation to copy a frame in the application in which you are working. Use the Import operation to copy a frame from another application. You must have rights to the application from which you are copying the frame.

  Both the Duplicate and the Import operations let you create frames with the same or similar queries but different frame types; for example, a Browse and an Update frame. Both operations are discussed in more detail in Copying an Existing Frame Definition (see page 101).
- Vision lets you include only tables that you have permission to use. However, you must make sure that the users of your application can access the tables you specify.
- You can include tables owned by another user, if that person has granted you permission. In this case, you specify a table by specifying *owner*.*tablename*. For more details about using the *owner*.*tablename* syntax, see the *4GL Reference Guide*. Be sure that the users of your application can also access the tables you specify.

Application Size Limits

Vision places these limits on the number of frames in an application:

- No frame can have more than 15 child frames.
- An application can have no more than 25 levels of frames, beginning with and including the top frame.

These limits apply to both new frames that you create and existing frames that you insert. A frame is counted once each time that you use it in an application.

Vision gives you an error message when you try to create or insert a frame that would cause your application to exceed these limits.

Create a New Frame

To create a new frame

1. Select a frame as the current frame.

   Vision always positions a new frame in relation to the current frame. If the current frame is the only frame, Vision automatically inserts a new frame below it. Otherwise, you must specify the position of the new frame. See step 2 to determine which frame to choose as the current frame.

2. Choose Create from the menu.

   Vision displays a list of locations:

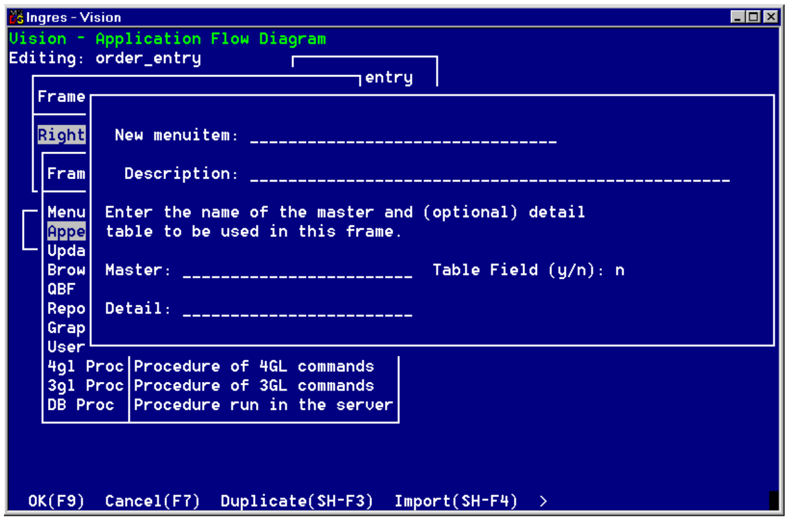
Location	Description
Left	Creates a peer to the left of the current frame.
Right	Creates a peer to the right of the current frame.
Down	Creates a child of the current frame. This choice is not available if the current frame already has children or cannot call other frames because it not a Vision-generated frame.

3. Position the cursor on a location and choose Select.

   Vision displays a list of the available frame types.

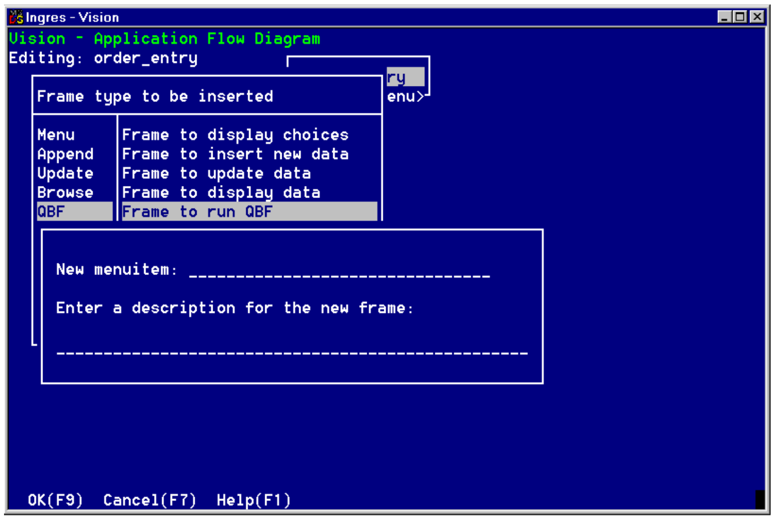
4. Position the cursor on a frame type and choose Select.

   Vision displays a pop-up window in which you enter basic information about the frame. The following figure illustrates this window for Append, Browse, and Update frames:



```
Ingres - Vision
Vision - Application Flow Diagram
Editing: order_entry
                                    entry
 Frame
 Right   New menuitem: ________________________________

  Fram     Description: __________________________________________________

  Menu   Enter the name of the master and (optional) detail
  Appe   table to be used in this frame.
  Upda
  Brow   Master: ________________________  Table Field (y/n): n
  QBF
  Repo   Detail: ________________________
  Grap
  User
  4gl Proc Procedure of 4GL commands
  3gl Proc Procedure of 3GL commands
  DB Proc  Procedure run in the server

 OK(F9)  Cancel(F7)  Duplicate(SH-F3)  Import(SH-F4)  >
```

The following figure illustrates this window for frame types that do not have visual queries—Menu frames and frames for which Vision does not generate the code:



```
Ingres - Vision
Vision - Application Flow Diagram
Editing: order_entry
                                               ry
 Frame type to be inserted                    enu>

 Menu      Frame to display choices
 Append    Frame to insert new data
 Update    Frame to update data
 Browse    Frame to display data
 QBF       Frame to run QBF

   New menuitem: _________________________________

   Enter a description for the new frame:

   __________________________________________________

  OK(F9)  Cancel(F7)  Help(F1)
```

5. Enter the text of the menu item that calls this new frame.

6. Press Tab to move the cursor to the description field.

7. Enter a description of up to 60 characters for the frame.

   Use care in specifying this description; it appears:

   - On any Menu frames that call this frame
   - As the title that Vision places at the top of the frame's form
   - In various help files associated with this frame

8. Proceed as follows depending on the type of frame you are creating:

   - For Menu frames, select OK.

     This is all you must do to create a Menu frame. If you want the menu items to be displayed in a style other than the application-wide default, see Specifying Menu Frame Display Styles (see page 113).

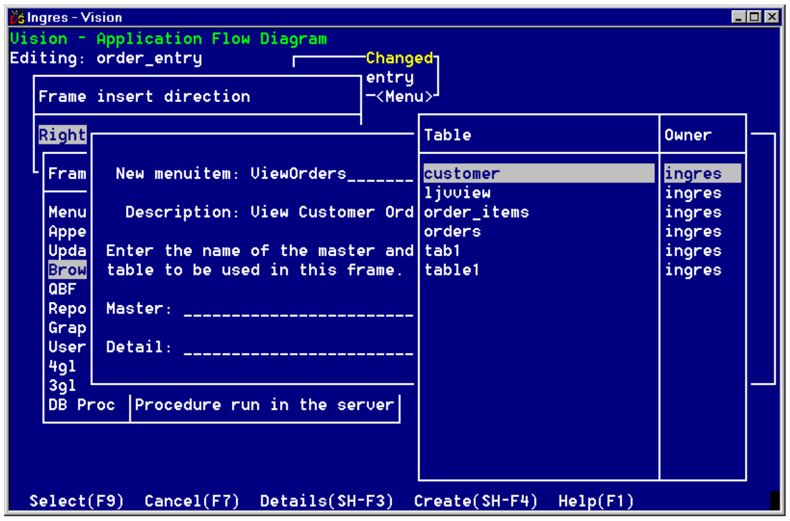
   - For Append, Browse, and Update frames, complete the steps below to specify the tables for the frame's visual query.

   - For frames for which Vision does not generate the code (for example, Report or QBF frames), see Defining Frames without Visual Queries (see page 159).

9. Press Tab to move the cursor to the Master field.

10. Specify the Master table in one of these ways:

    - Enter the name of a table that exists in the database.
    - Select ListChoices to see a list of the tables in your database that you have permission to use. This figure shows an example of such a selection list:



```
Ingres - Vision
Vision - Application Flow Diagram
Editing: order_entry                   Changed
                                       entry
  Frame insert direction               <Menu>

  Right                                   Table              Owner
   Fram   New menuitem: ViewOrders_______  customer           ingres
                                           ljuview            ingres
   Menu    Description: View Customer Ord  order_items        ingres
   Appe                                    orders             ingres
   Upda   Enter the name of the master and tab1               ingres
   Brow   table to be used in this frame.  table1             ingres
   QBF
   Repo   Master: ________________________
   Grap
   User   Detail: ________________________
   4gl
   3gl
   DB Proc  Procedure run in the server

  Select(F9)  Cancel(F7)  Details(SH-F3)  Create(SH-F4)  Help(F1)
```

    To choose a table listed, position the cursor on the name of the table and choose Select. Vision places the table name into the Master Table field.

    To display information about any of the tables listed, choose Details. This displays the Examine a Table window, which is described in Examining Tables (see page 245).

    - Create a new table to use with this frame by selecting ListChoices as above, then selecting Create from the submenu.

    Vision displays the Create a Table window in which you can create the new table. This window is identical to the table creation window in the Tables Utility. See the *Character-based Querying and Reporting Tools User* Guide for details.

    After you create the table, select Save to save your new table definition. Then select End to return to Vision.

    - Use the Duplicate or Import menu operation.

11. Press Tab to move to the Table Field (y/n) field.

12. Indicate whether you want to display the Master table on the form as simple fields or a table field. The default is simple fields.

    - To display the Master table as simple fields, go to the next step.
    - To display the Master table as a table field, type y. Then go to step 14. You cannot enter a Detail table name if the Master table is displayed as a table field.

13. To enter a Detail table, press Tab to move the cursor to the Detail table field. Enter the name of a table using any of the methods described in step 10.

    If you do not need a Detail table, go to the next step.

14. Select OK to save your responses.

    Vision verifies that you have permission to use the tables you designated. (When you select a table with the ListChoices operation, Vision only shows you the names of tables that it has already verified.)

    Vision displays the visual query window for the new frame. This window is discussed in Default Frame Definitions (see page 103).

Copying an Existing Frame Definition

You can create a new Append, Browse, or Update frame with the same frame definition as an existing frame. The frame that you are copying must also be an Append, Browse, or Update frame. However, the frames can be different types. For example, you can create an Append frame and copy the frame definition from a Browse frame.

When you copy an existing frame definition, Vision uses the tables and all visual query specifications for the existing frame. This includes any escape code that you have written for the frame, as described in Using Vision Advanced Features (see page 177).

You can keep this definition for the new frame or modify it with the Visual Query Editor, as described in Defining Frames with Visual Queries (see page 117).

You can copy the frame's form as part of the frame definition, or you can have Vision regenerate a default form.

Copy a Frame Within the Current Application

To copy an existing frame definition to a new frame

1. Follow steps 1 through 9 in Create a New Frame (see page 98) to create a new Append, Browse, or Update frame.

   At this point, Vision displays the table specification pop-up window.

2. With the cursor on the Master field, select Duplicate from the menu.

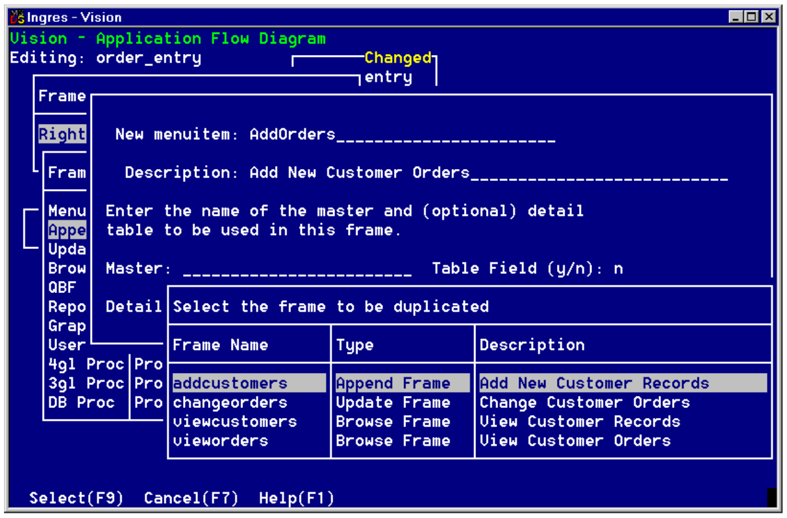
   Vision displays a selection list of all the Vision-generated frames that exist for the application, as shown in the following figure.

3. Position the cursor on a frame name and choose Select.

   Vision displays a pop-up asking whether you want to duplicate the form for the frame.

4. Select yes to specify that Vision should copy the frame's form. Select no to specify that Vision should regenerate a default form for the frame.

   Vision displays the visual query window for the new frame. Default Frame Definitions (see page 103) discusses this window:



```
Ingres - Vision
Vision - Application Flow Diagram
Editing: order_entry                  ┌──────Changed┐
                                                   entry
Frame
Right   New menuitem: AddOrders_______________________

Fram     Description: Add New Customer Orders___________________________

Menu  Enter the name of the master and (optional) detail
Appe  table to be used in this frame.
Upda
Brow  Master: ________________________  Table Field (y/n): n
QBF
Repo  Detail Select the frame to be duplicated
Grap
User         Frame Name         Type            Description
4gl Proc Pro
3gl Proc Pro addcustomers       Append Frame    Add New Customer Records
DB Proc  Pro changeorders       Update Frame    Change Customer Orders
             viewcustomers      Browse Frame    View Customer Records
             vieworders         Browse Frame    View Customer Orders

  Select(F9)  Cancel(F7)  Help(F1)
```

Copy a Frame from Another Application

To copy an existing frame definition from another application to a new frame

1. Follow steps 1 through 9 in Create a New Frame (see page 98) to create a new Append, Browse, or Update frame.

   At this point, Vision displays the table specification pop-up window.

2. With the cursor on the Master field, select Import from the menu.

   Vision displays a selection list of all the Vision applications that you have access to.

3. Position the cursor on the application name and choose Select.

   Vision displays a selection list of all the Vision-generated frames that exist for the application.

4. Position the cursor on the frame name and choose Select.

   Vision displays a pop-up asking whether you want to duplicate the form for the frame.

5. Select yes to specify that Vision copies the frame's form. Select no to specify that Vision should regenerate a default form for the frame.

   Vision displays the visual query window for the new frame. Default Frame Definitions (see page 103) discusses this window.

How You Can Change a Frame's Type

Duplicating a frame definition is a convenient way to change a frame's type. For example, to change a Browse frame to an Update frame, create a new Update frame and duplicate the definition of the original Browse frame. You can then remove the Browse frame from the application flow diagram and insert the new Update frame in its place. For more information, see Removing Frames (see page 108) and Insert Existing Frames (see page 109).

Using Default Frame Definitions and Forms

After you create an Append, Browse, or Update frame, Vision displays the visual query window for the frame. Because at this point you are using the specifications that Vision creates automatically, the query that Vision generates from these specifications is called the *default frame definition*.

Vision also generates a form based on the frame type and frame definition. For examples of the following, see Examples of Vision-generated Frames (see page 68).

- The default visual queries for Append, Browse, and Update frames
- The default form generated for Append, Browse, Update and Menu frames

This section describes the default frame definitions and forms that Vision generates.

Default Frame Definitions

The default frame definition for an Append, Browse or Update frame contains the basic specifications that Vision needs to run the frame and to generate:

- The code for the frame, including:
  - The database query for the frame
  - The menu items to appear on the frame
- The form associated with the frame

The visual query window containing the default frame definition displays:

- The columns in the Master table
- The columns in the Detail table, if you have specified one
- The *type* of the Master and Detail tables (the table type corresponds to the frame type—Append, Browse, or Update)

The visual query window also displays these specifications of the default frame definition:

- The join columns between the Master and Detail tables

  Vision draws a line between the first *natural join* (columns with the same name, data type and length) shared by the tables.

- The columns to include as fields on the frame's form and to use in the frame's database query

  By default, Vision marks all columns with a "y" in the Display on Form field to indicate that it should generate fields on the form for these columns and include them in the query.

  The only columns not marked as "y" are the join columns in the Detail table. You cannot display these columns as fields on the form, because they are duplicates of the same columns in the Master table.

- For Update frames, the default rules for inserting new records and deleting records. By default, Vision allows:
  - Insertions in the Detail table
  - Deletions in both the Master and Detail tables

Vision also sets the frame behaviors for Browse and Update frames as part of the default frame definition. The default frame behaviors are:

Qualification Processing

When a user runs the frame, the frame appears with a blank form; the user then can enter a query qualification to retrieve selected records.

Next Master Menu Item

Vision generates a Next menu item to let the user retrieve each record from the Master table (and corresponding records from the Detail table, if any) that satisfy the frame's query conditions.

The Next Master Menu Item frame behavior is not available if you have specified that the Master table appear as a table field.

For a description of how to modify any of the default frame definition specifications, see Defining Frames with Visual Queries (see page 117).

Default Transaction Processing

Vision-generated frames run in autocommit off mode. In Append frames, and Browse and Update frames that allow user qualifications, Vision inserts the following statements in the initialize block of the generated 4GL code:

```
commit work:
set autocommit off;
```

These statements are run before the form is displayed and before any Form-Start escape code is run. (For details on escape code, see Using Vision Advanced Features (see page 177).)

If an error occurs during a query when a generated frame is run, a rollback statement in the generated code causes any changes made since the last commit to be backed out.

Default Forms

For each frame that it generates, Vision also generates a default form. This form is based on the frame type and default frame definition.

Default Forms for Menu Frames

Menu frames call other frames, but do not have a query associated with them. The default form for a Menu frame includes:

- The frame name at the top
- The "Menu Frame" type in the upper right
- A table field (called "iitf" in the generated code) with two columns:
  - The first column is called "command" and contains the menu item text to call the child frames.
  - The second column is called "explanation" and contains the description of the child frames.

  The "command" table field column is scrollable, so that users can see long menu item text. Both table field columns are restricted; you cannot delete or rename them.
- A table field row for each child frame. To specify that the child frames be accessed by selecting from the menu line at the bottom of the window instead of from a table field, see Specifying Menu Frame Display Styles (see page 113).

Note: You can write escape code (discussed in Using Vision Advanced Features (see page 177)) to add commands to the table field, or to remove commands that you do not want users to see.

When you run the Menu frame, Vision generates a menu line containing Select, Help, and End menu items. Also, all the table field operations are available. For descriptions of these menu items and operations, see Vision Applications from a User's Perspective (see page 297).

Default Forms for Append, Browse, and Update Frames

The default forms that Vision generates are similar for Append, Browse, and Update frames. The default form for these frame types includes:

- The short description of the frame at the top
- The frame type in the upper right
- The displayed columns of the Master table represented as simple fields or as table field columns

If you have specified a Detail table in the frame definition, the default form also includes a table field with a column for each displayed column of the Detail table.

When it is run, the frame also includes a menu line containing:

- Each menu item that you specified to call another frame
- The Vision-generated menu items appropriate to the frame type and frame definition

For descriptions of the menu items that Vision generates, see Vision Applications from a User's Perspective (see page 297).

Attributes of Fields on Default Forms

Vision assigns attributes to fields on the default form as follows:

- You cannot display the join field in the Detail table.
- Any field in the Master or Detail table that activates a Lookup table is created as a mandatory field (that is, the user must make an entry in this field).
- Lookup table columns that you display as fields on the form are:
  - Query-only for simple fields
  - Display-only for table fields
- Sequenced fields in Append frames are created as query-only.

- Columns in a Master or Detail table that are not null not default are created as mandatory fields.
- All Master and Detail table columns that you include in the visual query are created as restricted fields; that is, you cannot:
  - Change the field's internal name
  - Change the field's data type
  - Delete the field from the form

The default form can change when you modify the frame definition. You can also use Vision's forms editor to modify a Vision-generated form. For more information on using the Vision forms editor, see Editing a Form (see page 153).

Appearance of Fields on Default Forms

Vision uses these visual signals to highlight special fields on the default form:

- The join field between the Master and Detail table appears in bold in the Master table.
- Any field in the Master or Detail table that activates a Lookup table is displayed in reverse video.

After Displaying the Default Frame Definition

When Vision displays the default frame definition, these options are available:

- Select End to save the default frame definition and return to the Application Flow Diagram Editor.

  You then can:

  - Continue to build the application flow diagram, or
  - Test the frame (as described in Completing a Vision Application (see page 259))
- Modify the default frame definition by using the Visual Query Editor. For more information, see Defining Frames with Visual Queries (see page 117).

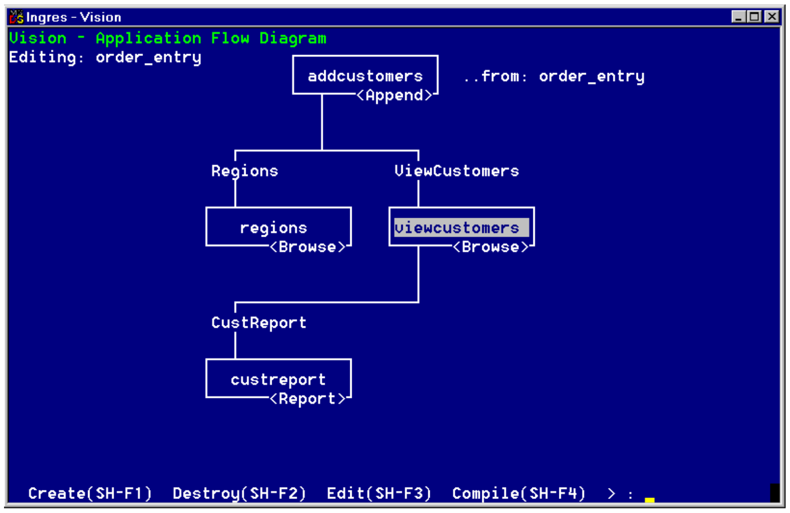
Removing Frames

When you remove a frame from the application flow diagram:

- Vision deletes the menu item that calls it from the parent frame.
- Vision removes all frames below the frame in its tree.
- The frame is not destroyed. You can put it back later, as described in the next section of this chapter.

If you later insert a frame back into the application—either the frame that you removed explicitly or another frame that was removed as part of the tree—Vision also inserts any children of that frame.

For example, in the application flow diagram in the following figure, if you remove the frame "addcustomers," Vision also removes the frames "regions," "viewcustomers," and "custreport."



```
Vision - Application Flow Diagram
Editing: order_entry
                          addcustomers         ..from: order_entry
                              <Append>

               Regions                ViewCustomers

                 regions              viewcustomers
                   <Browse>              <Browse>

               CustReport

                 custreport
                   <Report>

  Create(SH-F1)  Destroy(SH-F2)  Edit(SH-F3)  Compile(SH-F4)  > :
```

You then could insert these frames as follows:

- If you insert "addcustomers," Vision inserts all the frames that were removed.
- If you insert "viewcustomers," Vision also inserts "custreport" below it.
- If you insert "regions" or "custreport," no other frames are inserted, because neither of these frames has any children.

Vision does not insert removed frames that are at a higher level than the frame that you insert. For example, when you insert "viewcustomers," Vision does not also insert "addcustomers."

Remove a Frame

To remove a frame from the application flow diagram

1. Select the frame as the current frame.
2. Select Remove.

   Vision asks you to confirm that you want to remove the frame.
3. Select yes.

Vision removes the frame and any frames below it in its tree from the application flow diagram.

Insert Existing Frames

When you insert a frame that already exists in the application, you also insert all the frames below it in its tree.

Inserting frames is a convenient way to use a frame more than once in an application. For example, in the application flow diagram in the preceding figure, if you insert "viewcustomers," "custreport" is also inserted with it. However, you cannot insert a frame below itself in a tree, because this would result in a frame calling itself. Therefore you could not insert "addcustomers" as a child of "viewcustomers."

To insert an existing frame into the application flow diagram

1. Select the current frame adjacent to the frame to be inserted. See step 2 to determine which frame to choose as the current frame.
2. Select Insert from the menu.

   Vision asks where to insert the frame relative to the current frame—Right, Left, or Down.
3. Move the cursor to the position you want and choose Select.

   Vision displays a list of all frames in the application.
4. Move the cursor to the frame you want to insert and choose Select.

   Vision displays a pop-up window that prompts you for menu item text.
5. You can enter the frame's original menu item text or specify new text. If you are inserting the frame at the same level as the existing frame, you must specify new text.

   Vision inserts at the point you specified the frame and any lower-level frames of its tree.

Destroy Frames

Use the Destroy operation to completely delete a frame from the application. When you destroy a frame, you cannot later insert it into the application or perform any other operations on it.

Destroy a frame only if you are sure that you do not want to use it again; for example, if you specify the wrong tables or forget to include a Detail table.

When you destroy a frame, Vision:

- Removes any lower level frames in its family, but does not destroy them. You can insert any of these frames as described earlier in this chapter.
- Destroys any local variables and escape code (described in Using Vision Advanced Features (see page 177)) that you have created for the frame.

To destroy a frame

1. Select the frame as the current frame.
2. Select Destroy.

   Vision asks you to confirm that you want to destroy the frame.
3. Select yes.

   Vision destroys the current frame.

   Any frames below it in its tree are removed (but not destroyed) from the application flow diagram. To destroy any of these frames, repeat the above procedure.

Change Menu Item Text

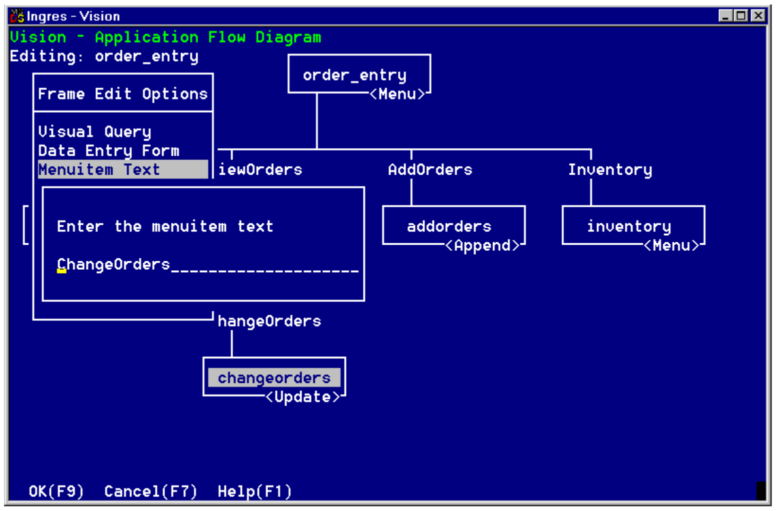
After you create a frame, you can change the text of the menu item that calls it. Vision changes the text on the parent frame and everywhere else it appears in the application.

To change the menu item text for a frame

1. Highlight the frame as the current frame.
2. Select Edit from the menu.
3. Select Menuitem Text from the list of Frame Edit Options.

   Vision displays a pop-up window containing the old menu item text, as shown in the following figure.
4. Type over the old text to enter the new menu item.
5. Select OK.

The new menu item text appears above the frame in the application flow diagram:



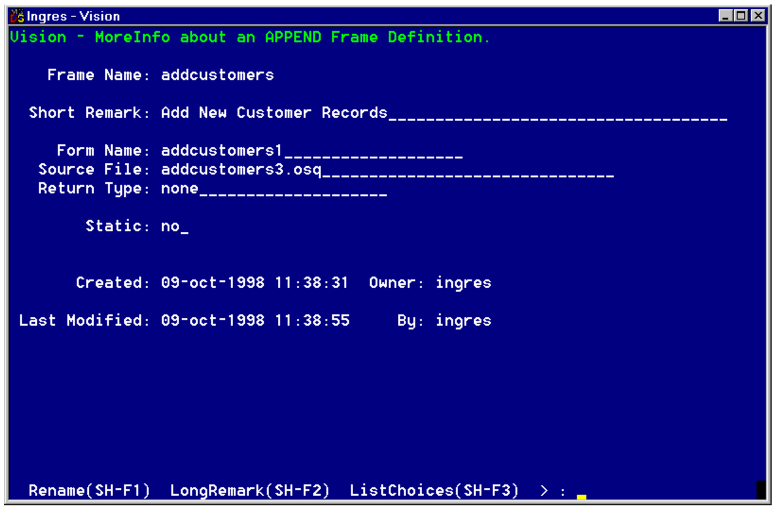
```
Ingres - Vision
Vision - Application Flow Diagram
Editing: order_entry
                                     order_entry
 Frame Edit Options                       <Menu>

 Visual Query
 Data Entry Form
 Menuitem Text      iewOrders         AddOrders          Inventory

   Enter the menuitem text            addorders          inventory
                                         <Append>            <Menu>
   ChangeOrders_____________________

                    hangeOrders

                      changeorders
                          <Update>

 OK(F9)  Cancel(F7)  Help(F1)
```

Note: The name of the frame remains unchanged. This protects any references to this frame from elsewhere in the application.

Viewing and Editing Frames

When you choose the MoreInfo menu item from the Application Flow Diagram Editor menu, Vision displays the MoreInfo about a Frame window. This window contains detailed information about the current frame.

The following figure shows the information for an Append frame:



```
Ingres - Vision
Vision - MoreInfo about an APPEND Frame Definition.

    Frame Name: addcustomers

  Short Remark: Add New Customer Records____________________________________

     Form Name: addcustomers1___________________
   Source File: addcustomers3.osq________________________________
   Return Type: none____________________

        Static: no_


       Created: 09-oct-1998 11:38:31  Owner: ingres

 Last Modified: 09-oct-1998 11:38:55     By: ingres



  Rename(SH-F1)  LongRemark(SH-F2)  ListChoices(SH-F3)  > :
```

You can view this information and, in some cases, change it. The following table describes the fields on the MoreInfo about a Frame window:

Field Name	Description	Changeable?
Frame Name	The name Vision gave the frame when you created it.	Yes To rename a frame, use the Rename operation on this window. The menu item and form for the frame are not affected.
Short Remark	The description that you have entered.	Yes
Form Name	The form associated with this frame.	Yes You can specify to use with this frame an existing form that you have created, rather than having Vision generate a default form (for details on sharing forms among frames, see Defining Frames with Visual Queries (see page 117)).
Source File	The file that contains the 4GL code for this frame.	Yes If the file you specify is not in the current directory, include the full directory path.
Return Type	The data type of any value returned by this frame; the default is "none".	Yes
Static	Indicates whether the values for this frame should be cleared each time the frame is called; the default is "no".	Yes
Created	The date the frame was created.	No
Owner	The user who created this frame.	No
Last Modified	The latest date on which the MoreInfo About a Frame Definition window was modified.	No
By	The user who modified it.	No

Edit Fields

To change the value of any of the editable fields

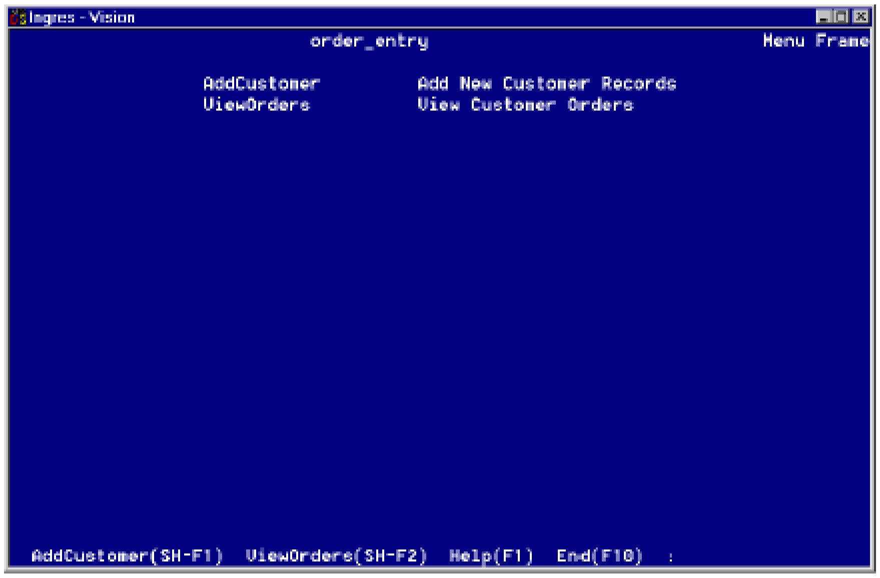
1. Tab to the desired field.
2. Enter the new value.
3. Repeat steps 1 and 2 for additional fields.
4. Select End to save your changes and return to the application flow diagram.

Specifyfing Menu Frame Display Styles

By default, Vision generates table-field Menu frames as described in Default Forms (see page 105). For an example of a table-field Menu frame, see the Menu Frame figure in Overview of Vision (see page 59).

You can also specify that Menu frames have *single-line menus* so users go to other frames by selecting a menu item from the menu line at the bottom of the window.

The following figure shows an example of a Menu frame with a single-line menu:



Ingres - Vision

```
                 order_entry                              Menu Frame

           AddCustomer          Add New Customer Records
           ViewOrders           View Customer Orders




 AddCustomer(SH-F1)  ViewOrders(SH-F2)  Help(F1)  End(F10)  :
```

You can specify that all Menu frames in an application be displayed with single-line menus or you can specify the single-line menu style on an individual frame basis. Each method is described below. The setting for a specific frame overrides the application-wide specification.

Specify Single-Line Menu Frames for an Application

When you create an application, you can specify that all Menu frames in an application be displayed with single-line menus. For more information on creating applications, see Overview of Tools and Languages (see page 35).

If you did not specify single-line menus when you created an application and want to change to that style, perform the following procedure.

To specify a single-line Menu frame

1. On the Applications Catalog window, position the cursor on the name of the application.
2. Select MoreInfo from the menu.

   The MoreInfo about an Application window is displayed.
3. Select Defaults from the menu.

   The Application Defaults pop-up window is displayed.
4. Use the Tab key to move the cursor to the Style for New Menu Frames field.
5. Type single-line.

   You can also use the ListChoices operation to specify the menu style.
6. Select OK from the menu.

   Vision asks you whether you want to change all existing Menu frames to single-line menus.
7. Press Return for "no" or move the cursor to yes and press Return.
8. Select Save.
9. Select End to return to the Applications Catalog window.

Specify a Single-Line Menu Style for a Frame

To specify the single-line style for an individual Menu frame

1. Select the Menu frame as the current frame in the Application Flow Diagram.
2. Select MoreInfo from the menu.

   The MoreInfo about a Menu Frame Definition window is displayed.
3. Use the Tab key to move the cursor to the Style for New Menu Frames field.

4. Type single-line.

   You can also use the ListChoices operation to specify the menu style.

5. Select End.

Editing Table-Field Menu Frames

You can make the following changes to table fields on Menu frames:

- Change the width of either the command or explanation table field column
- Add displayed or hidden columns to the table field

For more information about using the Vision forms editor, see Defining Frames with Visual Queries (see page 117).

Create Pop-up Menu Frames

You can use the following method to create Menu frames as pop-up windows on Append, Browse, or Update frames. When a user presses the Menu key on such a frame, the menu items for the child frames are displayed as a pop-up table field.

To create a pop-up menu frame

1. Create a Menu frame as the only child of the Append, Browse, or Update frame on which you want the pop-up to appear.

   Use descriptive menu item text for this Menu frame, for example, "Commands."

2. Edit the form for the Menu frame to change its style to pop-up.
3. Create any children of the Append, Browse, or Update frame as children of the Menu frame instead.

Thus, when a user selects Commands from the parent frame, a pop-up is displayed with a table field containing the menu item text for the child frames.

Chapter 6: Defining Frames with Visual Queries

This section contains the following topics:

This chapter describes how to use the Visual Query Editor to modify the definitions of Vision frames that manipulate data—Append, Browse, and Update frames.

Use the Visual Query Editor to:

- Indicate the columns of the Master and Detail tables to display on the form for a frame
- Include Lookup tables and joins in your frame definitions
- Allow or prohibit various user actions
- Specify additional features based on the frame type

Use the operations on the Visual Query Editor menu to perform the functions described in this chapter. For a complete list of the Visual Query Editor menu operations, see Vision Applications from a User's Perspective (see page 297).

This chapter also describes the ways in which you modify the form that Vision generates from the frame definition. Even though Menu frames do not have database queries, Vision generates a form for a Menu frame.

Call the Visual Query Editor

Display the visual query for a frame by these methods:

- When you create a new Append, Browse, or Update frame, Vision automatically displays the visual query display window, as illustrated in Overview of Vision (see page 59).
- To modify the frame definition for an existing frame, explicitly call the Visual Query Editor.

To call the Visual Query Editor for an existing frame

1. Select the frame as the current frame in the application flow diagram.
2. Select Edit from the menu.

   Vision displays the Frame Edit Options list.
3. Select Visual Query from the list.

Vision displays the visual query window for the frame.

Viewing the Visual Query

Use these two ways to look at a visual query on your terminal window:

- When in *compressed view*, you can see the "big picture" of a visual query by selecting ZoomOut to be in this view.
- In *normal view*, you can define a frame and see the details of a visual query; this is the default view that Vision uses.

The sections that follow describe how and when to use each of these views.

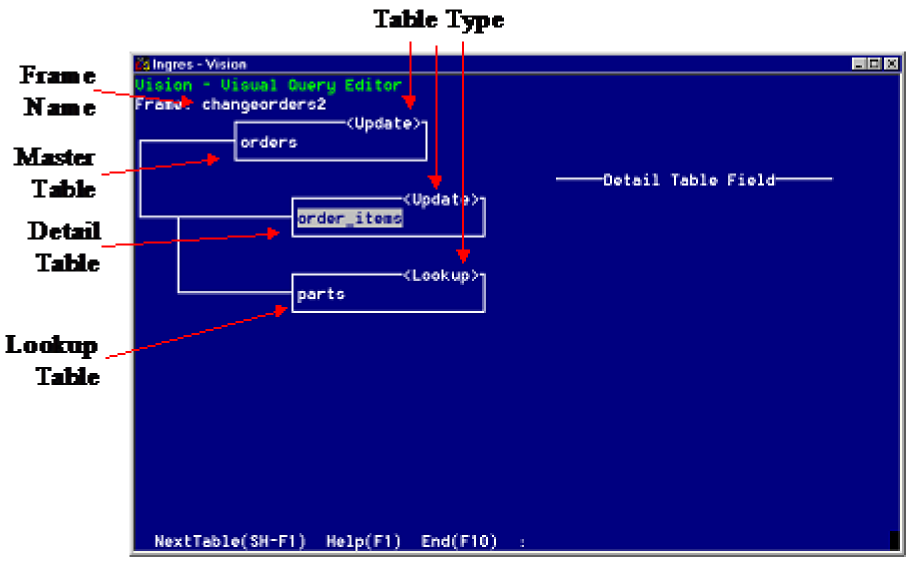
Compressed View

If your visual query has many tables and columns, Vision cannot display them all in one window. To see the overall frame definition, you must use the compressed view.

The compressed view usually fits on one window, because it lacks the detail of the normal view. The compressed view generally is adequate for seeing the basic structure of a query.

The compressed view shows you the names of the tables used in the query and their relationship to each other. You can see which tables are joined, but not the actual join columns. This view also indicates which are the Master and Detail tables.

The following figure shows a compressed view of a visual query:



In the compressed view:

- The frame name appears in the upper left of the window.
- Each box indicates a table used in the query.
- The box for a table contains the table name in the center and the table type (Append, Browse, Update, or Lookup) in brackets in the upper right corner.
- Lookup tables appear directly under the Master or Detail table with which they are associated.
- If you have specified that the Master table be displayed on the form as a table field, then the following line appears to the right of the Master table in the visual query display:

  ```
  "---Master Data in Table Field---"
  ```

- If the frame has a Detail table, then the Master table and any associated Lookup tables are above the line:

  ```
  "---Detail Table Field---"
  ```

  The Detail table and any associated Lookup tables are below the line.

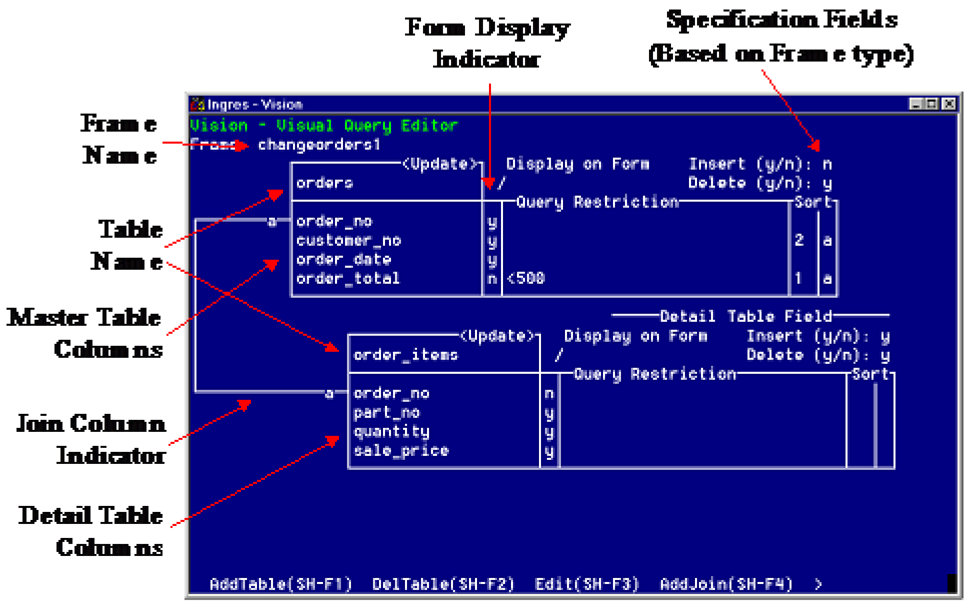
- Vision draws a line between tables that are joined.

  You cannot perform any operations on a visual query in compressed view. However, you can select the *current table*—that is, the table with which you are working—by using the NextTable menu item or the arrow keys.

  You then can return to normal view as described below to continue to edit the visual query.

Normal View

Work with the Visual Query Editor in normal view to specify or examine the details of a visual query. The following figure shows a portion of the same query as above represented in normal view:



```
Vision - Visual Query Editor
Frame: changeorders1
              <Update>  Display on Form    Insert (y/n): n
  orders                /                  Delete (y/n): y
                          Query Restriction              Sort
a order_no            y
  customer_no         y                                  2  a
  order_date          y
  order_total         n <500                             1  a

                                       Detail Table Field
                     <Update>  Display on Form    Insert (y/n): y
         order_items           /                  Delete (y/n): y
                                 Query Restriction              Sort
       a order_no            n
         part_no             y
         quantity            y
         sale_price          y

 AddTable(SH-F1)  DelTable(SH-F2)  Edit(SH-F3)  AddJoin(SH-F4)  >
```

In normal view, the visual query display is expanded so that:

- Vision shows the names of all the columns in the Master and Detail tables and any Lookup tables that you include (only the first 19 characters of each column name are displayed).
- The Master table and Detail table (if the frame has one) are represented as table fields in the visual query display. In each table field, one row corresponds to a column of the database table.
- Vision draws lines between the specific columns on which the tables are joined. The join lines are designated in alphabetic order; that is, the first join is marked "a," and so on.
- The Display on Form field shows which columns are displayed as fields on the form.
- For each column, Vision provides a field in which you can specify query restrictions (for Browse and Update frames) or assign values (for Append frames).
- For Update frames, there are simple fields that indicate whether the user can append and delete data to the Master table and Detail tables.

Move Between Normal and Compressed View

When you call the Visual Query Editor, Vision puts you into normal view.

To move between the compressed view and normal view of a visual query

1. Select ZoomOut from the menu.
2. Use the up and down arrow keys to select the current table in compressed view.
3. Select End to return to normal view.

   The cursor is positioned on the first field of the current table.

Moving Around in the Visual Query Editor

To move around the Visual Query Editor, you use:

- The up and down arrow keys to scroll between rows in a table field
- The Tab key to move:
  - Forward through the columns of a table field row
  - To the first simple field for each table—for example, the Insert (y/n): field shown in the preceding figure—from the last column in any row of the table field
  - Forward between simple fields
- The Previousfield key to move:
  - Backward between columns of a table field row
  - Backward between simple fields
  - To the first row of the previous table field (from the first column of any row in a table field)
- The NextTable operation to move the cursor to the first field of the next table in the visual query display

To see which keys to use in the visual query display window, select Help from the menu, then select Keys from the submenu. In particular, you can use this function to find out the keys to which the Previousfield function is mapped for your PC or terminal.

Change the Default Form Display

By default, Vision creates a field on a frame's form for each column (or table field column) in the Master and Detail table. Vision also includes each column as part of the frame definition when it generates the code for a frame.

There are two restrictions on displaying columns:

- You cannot display the join columns in the Detail table, because they are duplicates of the join columns in the Master table.
- You must display the column that joins the Master or Detail table to the Lookup table—although you cannot display the join column of the Lookup table.

  If you do not want the user to see this column, you can edit the form to make the column invisible. See Modifying Vision-generated Forms (see page 152) for more information on editing forms.

Besides the default action, you have two additional options for specifying the display of columns:

- If you do not want to display a particular column on the form or include it as part of the query, mark that column as n (do not display) on the visual query display window.

  When Vision generates the code and the form for the frame, it ignores any columns marked as n (except in certain cases on Append frames, as discussed in Defining Append Frames (see page 134)).
- If you want to include a column in the generated code but not display it on the form, mark that column as v (create a local variable) on the visual query display window. Vision creates a local variable based on this column that you can use in several ways, as discussed in Using Vision Advanced Features (see page 177).

  The value of the local variable is determined by the values of the current row of the database table of which its column is part. When these values are changed (by a user selecting the next Master table record, for example), the value of the local variable changes along with the values for the displayed columns of the table.

To change the default display of a column (or table field column)

1. Position the cursor on the name of the column in the visual query display.
2. Tab to the Display on Form field.
3. Type n or v.

In the preceding figure, the "order_total" column of the Orders table has been marked as n for a non-displayed field.

Redisplay a Column

When you redisplay a column that you have marked as nondisplayed or made a local variable, Vision places the field for that column at the top of the form. You must edit the form to move the column to a new location on the form.

To redisplay a column as a field on the form

1. Position the cursor on the name of the column in the visual query display.
2. Tab to the Display on Form field.
3. Type y in the Display on Form field.

Including Lookup Tables in Visual Queries

A visual query can contain no more than one Master table and one Detail table. However, you can designate other tables to use as *Lookup tables* in your frame definitions.

Use Lookup tables to:

- Let the user select an appropriate value for a field

  If you include a Lookup table for a field and the user selects the ListChoices operation, Vision displays a selection list containing the valid choices. This allows the user to select an item rather than typing a value.

- Validate an entry that the user makes

  If you include a Lookup table for a field, Vision validates a user's entry against the Lookup table. The user does not need to use the ListChoices operation.

  If the user enters an invalid value, Vision displays an error message that tells the user that the ListChoices operation is available.

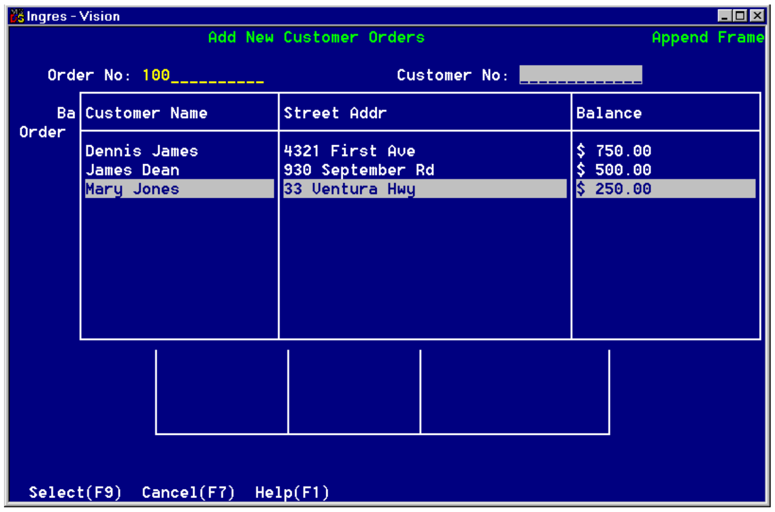
- Allow Vision to enter values directly into specified fields on the form

  In the visual query, you can specify fields on the form—in addition to the field that activates the Lookup table—into which Vision inserts a value. Vision enters the appropriate value when the user chooses an item from the selection list.

  If you prohibit the user from entering query restrictions on a Browse or Update frame (as discussed in Qualification Processing (see page 146)), Vision enters data into the form fields without displaying a selection list. Vision displays the form with values already entered, including any values that you specify from the Lookup table.

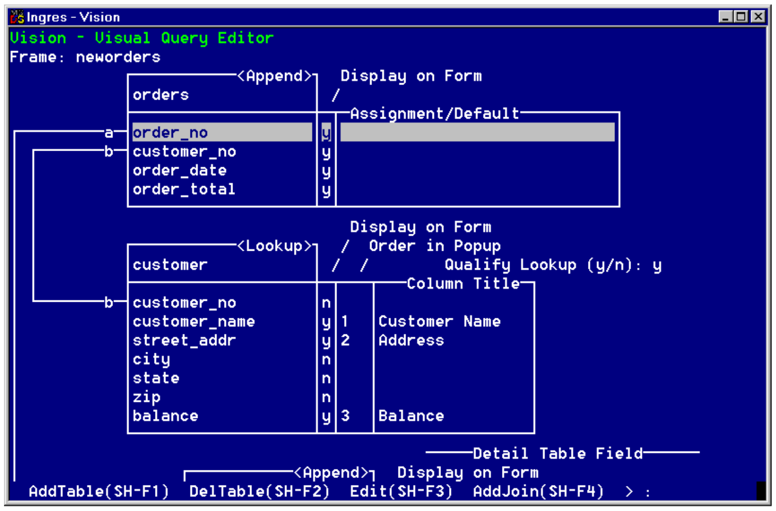
As an example of a typical use for a Lookup table, assume an order entry clerk uses an Append frame for new orders. The clerk must enter a customer number for each order. The clerk knows a customer's name but not the customer's number. To help the clerk, you can include the Customers table as a Lookup table in the visual query.

When the clerk selects ListChoices on the customer number field, Vision displays a selection list that provides customer names, and even addresses and account balances:



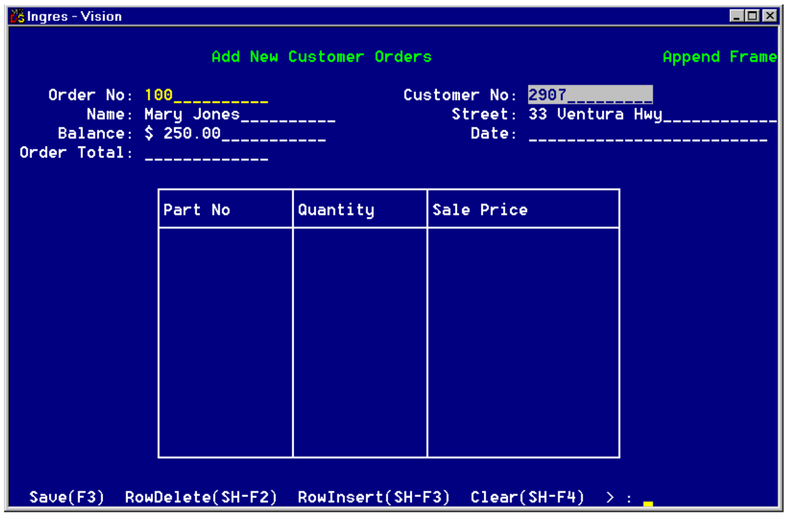
```
Ingres - Vision
                        Add New Customer Orders                    Append Frame

    Order No: 100__________                   Customer No: _____________

     Ba| Customer Name       | Street Addr                   | Balance
  Order|                     |                               |
       | Dennis James        | 4321 First Ave                | $ 750.00
       | James Dean          | 930 September Rd              | $ 500.00
       | Mary Jones          | 33 Ventura Hwy                | $ 250.00

  Select(F9)  Cancel(F7)  Help(F1)
```

Here is the visual query used to generate the Lookup table:



```
Ingres - Vision
Vision - Visual Query Editor
Frame: neworders
             ┌──────────<Append>┐  Display on Form
             │orders            │ /
             │                  ├─Assignment/Default──────────┐
 ┌─────────a─│order_no          │y│                           │
 │  ┌──────b─│customer_no       │y│                           │
 │  │        │order_date        │y│                           │
 │  │        │order_total       │y│                           │
 │  │        └──────────────────┴─┴───────────────────────────┘
 │  │                               Display on Form
 │  │        ┌──────────<Lookup>┐  /  Order in Popup
 │  │        │customer          │ /  /        Qualify Lookup (y/n): y
 │  │        │                  ├─┬─┬────Column Title─┐
 │  └──────b─│customer_no       │n│ │                 │
 │           │customer_name     │y│1│ Customer Name   │
 │           │street_addr       │y│2│ Address         │
 │           │city              │n│ │                 │
 │           │state             │n│ │                 │
 │           │zip               │n│ │                 │
 │           │balance           │y│3│ Balance         │
 │           └──────────────────┴─┴─┴─────────────────┘
 │                                     ──────Detail Table Field──────
 │                 ┌──────────<Append>┐  Display on Form
  AddTable(SH-F1)  DelTable(SH-F2)  Edit(SH-F3)  AddJoin(SH-F4)  > :
```

When the user selects a name from the list, Vision enters the values from the selected item onto the form:



```
Ingres - Vision

                     Add New Customer Orders                      Append Frame

    Order No: 100__________             Customer No: 2907_________
        Name: Mary Jones__________           Street: 33 Ventura Hwy____________
     Balance: $ 250.00___________              Date: _________________________
 Order Total: _____________

                 | Part No      | Quantity     | Sale Price         |

  Save(F3)  RowDelete(SH-F2)  RowInsert(SH-F3)  Clear(SH-F4)  > :
```

Notice that the field that activates the Lookup table (the customer number) need not be displayed in the selection list.

The previous example is just one way to use a Lookup table in a visual query. For a detailed description of the ways in which users can use Lookup tables and the ListChoices operation, see Using Lookup Tables and the ListChoices Operation (see page 297).

Guidelines for Using Lookup Tables

Observe these guidelines when you use Lookup tables in your visual queries:

- Use Lookup tables with Append, Browse, or Update frames only.
- The column that activates the Lookup table must be displayed as a field on the form.

  Vision creates this field as mandatory—that is, the user must enter a value. You can edit the form to allow the user to skip over this field.
- Include a total of up to eight Lookup tables for a frame that has a Master and a Detail table.

  If your visual query does not use a Detail table, you can include nine Lookup tables.

- The Lookup table can share only one join column with the table for which you specify it; this is the column that activates the Lookup table.

  To activate the same Lookup table on more than one column, you must include the Lookup table in the visual query for each of the columns. Each occurrence of the Lookup table counts toward the total number of tables allowed.
- If the selection list for a Lookup table is too wide to display as a pop-up window, Vision displays it as a full window.
- Vision makes the Top, Find and Bottom scrolling operations available to the user to move through a selection list; however, these operations do not appear as menu items.
- The user cannot perform any data manipulation operations (for example, update or delete) directly on a selection list.

Specifying Features of Lookup Tables

After you include a Lookup table in a frame definition, you can specify these features:

- The columns of the Lookup table to display as fields on the form
- The columns to display on the selection list
- The order of the displayed columns on selection list
- The titles for the displayed columns on the selection list
- Whether users can enter query restrictions to specify records to be displayed in the selection list
- The join column between the Lookup table and the Master or Detail table; that is, the column that activates the Lookup

The remainder of this section describes how to create Lookup tables and specify these features.

Insert Lookup Tables

To include a Lookup table in a visual query

1. Position the cursor anywhere on the visual query display of the Master or Detail table to which you are joining the Lookup table.
2. Select AddTable from the menu.

   Vision displays a pop-up window that asks for the name of the Lookup table.

3. Provide the Lookup table name in either of these ways:
   - Enter the name of the table on the pop-up window.
   - Use the ListChoices operation to select a table.

     Vision enters the name of the table that you select onto the pop-up window.
4. Select OK.

Vision inserts the Lookup table on the visual query window just below the table to which it is joined. Vision also draws a line indicating the first *natural join* (columns with the same name, data type and length) shared by the two tables.

This is the column that activates the Lookup table. If you want to activate on a different column, you must change the join.

If Vision cannot find a natural join, it adds the Lookup table, but displays a warning message. You must create a join before you run the frame.

Adding and deleting joins are discussed elsewhere in this chapter.

Display Lookup Table Columns as Fields on the Form

Unlike Master and Detail tables used in queries, the default for Lookup tables is that a column not be displayed as a field on the form. If necessary, you can change this default to display Lookup table columns as fields on the form.

When the user selects an item from the selection list, Vision enters its values into the fields on the form that correspond to the specified columns of the Lookup table.

Note: Any data from the Lookup table that you include on the form is *display-only*; the user cannot manipulate their values.

You cannot display the join column of the Lookup table.

When the user selects Go on a frame with Lookup table columns displayed as form fields, Vision generates a select statement that retrieves data from the Lookup table and the Master or Detail table to which it is joined.

If a Lookup table column has the same name as the column in the Master table or Detail table to which it is joined, Vision:

- Adds a sequence number to the name of the Lookup table column in the visual query display
- Marks the column with the symbol "@"
- Does not change the field's title on the form, but does change the field's internal name to include the sequence number

For example, if a Master table has a column called Name and you display a column in its Lookup table called Name, the Name column in the Lookup table appears as Name1 in the visual query. If you edit the form, this field appears as Name, but with an internal name of Name1.

Also, Vision generates duplicate fields on the form for Lookup table and Master or Detail table columns with the same name. You can handle this situation in either of two ways:

- If the columns have the same name and contain the same data—for example, an Address column with customer addresses—leave the Lookup table column non-displayed. In this way, the customer addresses are displayed once only on the form.
- If the columns have the same name but contain different data—a Name column in the Parts table with part names and a Name table in the Customer table with customer names, for example—then edit the form to change the title of the form field for one of the columns.

You can tell which field relates to the Lookup table column because its internal name contains a sequence number, as described previously.

To display a column of a Lookup table as a form field

1. Position the cursor on the name of the column in the visual query display for the Lookup table.
2. Tab to the Display on Form field.
3. Type y.

Display Lookup Table Columns in the Selection List

By default, Vision displays the join column of the Lookup table on the selection list that the user sees after selecting ListChoices (except as described in How Using a Lookup Table Without Displaying a Selection List Works, in this topic). To include other columns in addition to or instead of the join column, you must tell Vision explicitly:

- Which columns of the Lookup table to display

  Except as noted below, your selection list must contain at least one column.

- The order in which to display the columns

  Vision uses the column display order as the order in which to sort the records that are retrieved from the Lookup table, also.

By default, Vision displays the title of a column on the selection list as it appears on the visual query display, but with the first letter changed to uppercase. For example, the "balance" column is displayed as "Balance."

You can display the column with a different title. In the visual query in the preceding figure, for example, the "street" column is displayed to the user with the caption "Address."

When you specify the display order with the procedures below, notice that Vision designates the join column (or first column, if there is no join column) as the first displayed column. To add other columns, begin your numbering with "2," rather than "1" as indicated.

To make a different column the first column, ignore the default and begin with "1" for the column you want to display first. You then can change the number for the join column or type a blank over it to remove it completely.

To specify the columns to display in the selection list

1. Position the cursor on the name of the column you want as the first displayed column in the selection list.
2. Press Tab.
3. Enter the number 1 in the Order in Pop-up field.
4. To accept the default column title, proceed to Step 5.

   To change the column title:

   a. Press Tab to move to the Column title field.

   b. Enter a new title.
5. To display additional columns, move the cursor to the row for the next column and repeat the above procedure. Enter the appropriate number (and column title, if desired) for each column.

If you change the display order for a column, Vision repositions all the subsequent columns.

To remove a column from the display, type a blank space in the Order in Pop-up field for that column. To change the column title, enter a new title over the current text.

How Using a Lookup Table Without Displaying a Selection List Works

You can define a Browse or Update frame so that users cannot enter query qualifications when the frame is displayed. (See Specifying Frame Behaviors (see page 145) for details.)

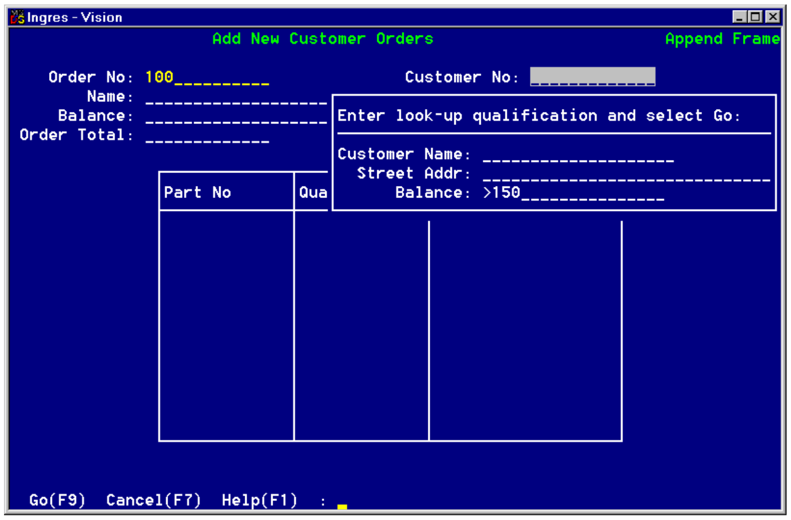
If you display columns of the Lookup table as fields on the form, Vision fills in the form based on the values from the first item in the Lookup table (or the first item that meets any query restrictions that you specify in the visual query).

Because the data already is entered into the form fields when the frame is displayed, there is no selection list from which users can choose. Also, Vision does not generate the ListChoices menu item for the frame.

Specify User Lookup Qualifications

You can let users enter qualifications so that only certain records of the Lookup table are retrieved. When a user selects ListChoices, Vision displays a pop-up window on which the user can enter a value for any of the columns in the selection list. If you allow user retrieval qualifications, they are permitted on all columns of the selection list.

For example, if the user enters ">150" into the Balance field, the selection list only displays those customers whose account balance is greater than $150. The following figure shows the pop-up window on which the user can enter a qualification:



```
Ingres - Vision
                    Add New Customer Orders                        Append Frame

    Order No: 100__________              Customer No: ____________
        Name: ___________________ 
     Balance: ___________________ Enter look-up qualification and select Go:
 Order Total: _____________
                                  Customer Name: ____________________
                                    Street Addr: ______________________________
               Part No       Qua        Balance: >150_______________

  Go(F9)  Cancel(F7)  Help(F1)  :
```

Using a qualification lets a user find a specific item quickly, especially if the Lookup table contains a large number of records. It also conserves memory on your system, because Vision only loads into the selection list the rows of the Lookup table that meet the user's qualifications.

The default is not to allow user qualifications. You must explicitly tell Vision to allow them.

To allow user qualifications

1. Tab to the Qualify Lookup field on the Lookup table visual query display.
2. Type y.

   To change the specification back to the default, type n in the Qualify Lookup field.

Specify Joins for the Lookup Table

By default, Vision joins the Lookup table to the Master or Detail table by the first natural join that they share. This is the column that activates the Lookup table.

If Vision does not find a join, you must create one before you can run the frame. To activate the Lookup table on a different column, join the Lookup table to the Master or Detail table on that column. You must remove the current join and then create a new one.

Creating and removing joins are described elsewhere in this chapter.

If you change the field on which the Lookup is activated, Vision highlights the new field and removes the highlight from the original field.

Remove Lookup Tables

To remove a Lookup table from a visual query

1. Position the cursor anywhere on the visual query display for the Lookup table.
2. Select DelTable from the menu.

   Vision displays a pop-up window that asks you to confirm that you want to delete the Lookup table.
3. Select yes.

   Vision removes the Lookup table from the frame definition and from the visual query display.

Inserting and Removing Joins

Vision uses the following rules to join tables in a visual query:

- Master and Detail tables are joined on each *natural join* (columns that share the same name, data type and length).

  This means that Vision can join columns that appear to be the same, but that really contain different information; for example, a store address in a Master table and a customer address in a Detail table.
- Lookup tables are joined to their associated Master or Detail table on the first natural join.

You use the Visual Query Editor to add or delete joins. You must specify the correct joins so that Vision generates the correct query for a frame. Also, the join column to a Lookup table determines the column that activates the ListChoices operation for users.

For Lookup tables, you first must remove the default join before specifying a new one.

The Visual Query Editor warns you if you try to exit without any joins specified between tables. You cannot compile or test the frame until you correctly specify the required joins.

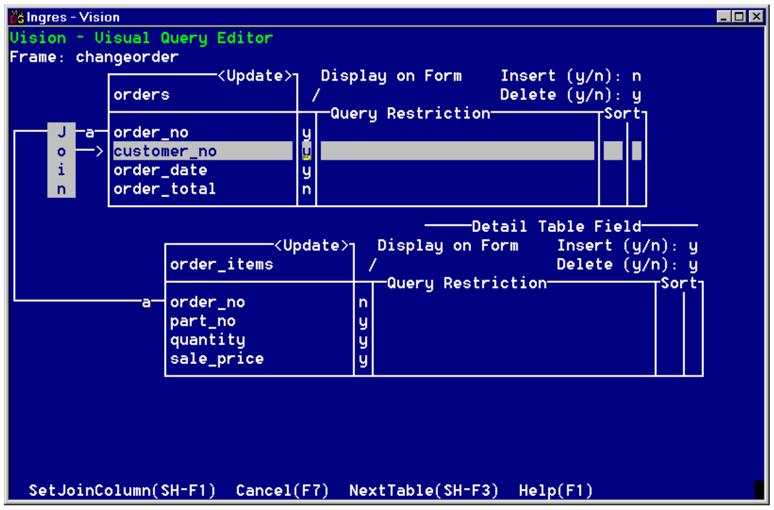
Insert a Join

To insert a new join in the visual query

1. Select AddJoin from the Visual Query Editor menu.
2. Position the cursor on the name of the join column in the first table that you want to join.

   You can join tables in any order; for example, Master to Detail table or Lookup table to Master table.
3. Select SetJoinColumn from the submenu.

   Vision marks this column as a join column:



```
Ingres - Vision
Vision - Visual Query Editor
Frame: changeorder
                 <Update>   Display on Form     Insert (y/n): n
           orders          /                    Delete (y/n): y
                              Query Restriction             Sort
     J  a  order_no       y
     o  -> customer_no    y
     i     order_date     y
     n     order_total    n

                                     Detail Table Field
                      <Update>   Display on Form     Insert (y/n): y
                order_items     /                    Delete (y/n): y
                                   Query Restriction            Sort
             a  order_no       n
                part_no        y
                quantity       y
                sale_price     y


  SetJoinColumn(SH-F1)  Cancel(F7)  NextTable(SH-F3)  Help(F1)
```

4. Move the cursor to the join column in the second table.
5. Select SetJoinColumn from the submenu.

   Vision draws a line between the columns in the Master and Detail table.

Remove a Join

To delete a join between two tables in the visual query

1. Position the cursor on the name of the join columns in either table.
2. Select DelJoin from the menu.

   Vision asks you to confirm the deletion.
3. Select yes.

   Vision removes the join.

Defining Specific Frame Types

Append, Browse, and Update frames let users manipulate data in a variety of ways. Your visual query specifications determine the operations that the user can perform on these frames.

The sections that follow describe how you use the Visual Query Editor to define Append, Browse, and Update frames. For a summary of the available user operations and the corresponding visual query specifications, see Vision Applications from a User's Perspective (see page 297).

Defining Append Frames

Append frames let users add records to tables. You can specify these features for the visual query of an Append frame:

- *Sequenced fields* for the Master table
- *Default values* for displayed columns
- *Assigned values* for non-displayed columns

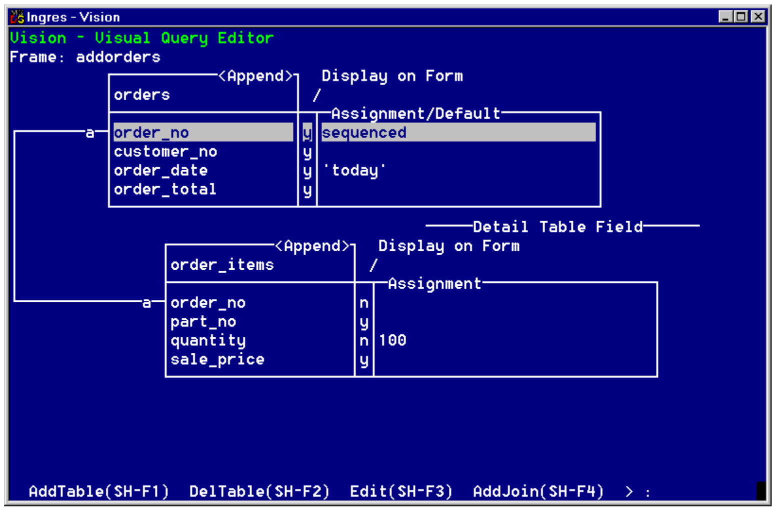
The figure in the following section shows a visual query window for an Append frame. The specifications in the figure correspond to the examples used in the discussion below.

Specify Sequenced Fields

To ensure that each record that users add on an Append frame has a unique value in the key field, you can specify that a column of the Master table appear as a *sequenced field* on the form. When a user calls the Append frame, Vision inserts a new integer value into the sequenced field.

Vision sets the value of the sequenced field to "1" for the first record to be added. Each time the user appends a record to the table, Vision increases the value of the sequenced field by one, thus ensuring that each record has a unique key.

For example, you can have an Append frame on which users add new orders. You can specify the order number column as a sequenced field on the form. Vision automatically generates a unique, sequential number for each new order.



```
Ingres - Vision
Vision - Visual Query Editor
Frame: addorders
                               <Append>   Display on Form
            orders                          /
                                           Assignment/Default
    a  order_no                   y  sequenced
       customer_no                y
       order_date                 y  'today'
       order_total                y

                                           Detail Table Field
                                  <Append>   Display on Form
                 order_items                   /
                                               Assignment
           a     order_no                    n
                 part_no                     y
                 quantity                    n  100
                 sale_price                  y

  AddTable(SH-F1)  DelTable(SH-F2)  Edit(SH-F3)  AddJoin(SH-F4)  > :
```

You can specify one column on an Append frame as the sequenced field. This column:

- Must be in the Master table
- Must be a key column of the table
- Can be either displayed or non-displayed
- Must have a data type of integer(4) to hold the value that Vision generates

You cannot specify a sequenced field if you have displayed your Master table as a table field.

Vision increments the value of the sequenced field as follows:

- If the sequenced field is displayed on the form and the user exits the frame without saving the record, Vision still increments the value of the field the next time the frame is called. Thus, it is possible to have "holes" in the sequence where records were not saved.
- If the sequenced field is not displayed, Vision only increments the value when the user selects Save. In this case, there are no holes in the sequence.

To specify that a column appear as a sequenced field

1. Position the cursor on the name of the column in the Master table.
2. Tab to the Assignment/Default field.
3. Type the word sequenced.

Set Default Values for Displayed Fields

You can indicate a *default value* for a column displayed as a form field on an Append frame. This default value is displayed in the field or assigned as the value of a local variable (for a column marked as v in the visual query display) each time the form is cleared for new use.

For example, if you want the Date column to always contain the current date without the user having to enter it. You can specify 'today' (a built-in Ingres function) as the default value for the Date field.

The field for which you specify a default value:

- Must be in the Master table
- Must be displayed as a field on the form
- Cannot already be assigned a value of "sequenced" (see Specify Sequenced Fields, in this topic)

If you have displayed your Master table as a table field, you cannot specify default values for columns of the table field.

To specify a default value for a Master table field

1. Position the cursor on the name of the column in the Master table.
2. Tab to the Assignment/Default field.
3. Enter a default value for the column.

   The value must be consistent with the column's data type. The value can be in the form of an expression as described in Using Expressions in Frame Definitions (see page 155).

Note: Be aware that the user can type a new value over the default value that Vision places into a field. To prevent this from happening, edit the form to make the field *display-only*. When the user moves through the form, the cursor skips over this field and the user cannot enter any data into it.

Assign Values to Non-Displayed Columns

You can specify a value to insert into a non-displayed column (that is, a column marked as n in the visual query display) of an Append frame. For each new record that you append, Vision enters the value directly into the column without it appearing on the form.

In the example in the preceding figure, the quantity for each item ordered automatically is set at 100. Users cannot see—and, therefore, cannot change—this value.

The non-displayed column to which you assign a value:

- Can be in the Master table or the Detail table
- Cannot already be assigned a value of "sequenced" (see Specify Sequenced Fields, in this topic)

To specify an assigned value for a non-displayed column

1. Position the cursor on the name of the column.
2. Change the Display on Form indicator to n.
3. Tab to the Assignment/Default column in a Master table or the Assignment column in a Detail table.
4. Enter a value for the column.

   The value must be consistent with the column's data type. The value can be in the form of an expression as described in Using Expressions in Frame Definitions (see page 155).

How You Can Control Window Clearing

By default, Append frames are cleared each time a user selects Save. You can prevent the window from being cleared by setting the internal 4GL variable "IIclear" to "n" in escape code.

For example, put the following statement in a Form-Start escape:

```
IIclear = 'n';
```

For instructions on how to write escape code, see Using Vision Advanced Features (see page 177).

Defining Update Frames

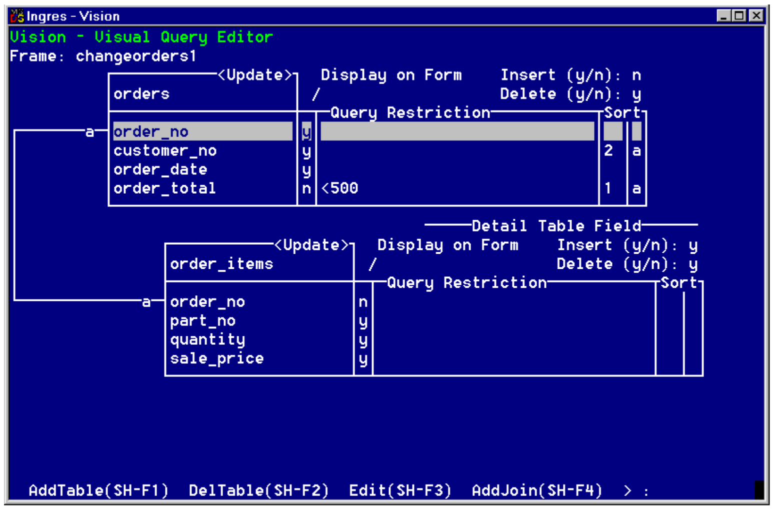
Update frames let the user retrieve records from tables, then update the retrieved data. You also can define an Update frame to include any of these additional operations:

- Inserting new records into the Master and Detail tables
- Deleting records from the Master and Detail tables
- Qualification processing to let users retrieve specific records
- A Next menu item to let users retrieve all Master table records that meet the query specification

You also can specify for an Update frame:

- The column sort order for retrieved records
- Query restrictions that you specify in the visual query to retrieve specific records (in addition to or instead of letting users enter query qualifications)

  The following figure illustrates the visual query window for an Update frame:



```
Ingres - Vision
Vision - Visual Query Editor
Frame: changeorders1
            ┌───────────<Update>┐  Display on Form    Insert (y/n): n
            │orders             │ /                  Delete (y/n): y
            ├───────────────────┼─┬─Query Restriction──────────┬Sort┐
   ┌──────a─┤order_no           │y│                            │  │ │
   │        │customer_no        │y│                            │2 │a│
   │        │order_date         │y│                            │  │ │
   │        │order_total        │n│<500                        │1 │a│
   │        └───────────────────┴─┴────────────────────────────┴──┴─┘
   │                                    ─────Detail Table Field──────
   │              ┌───────────<Update>┐  Display on Form    Insert (y/n): y
   │              │order_items        │ /                  Delete (y/n): y
   │              ├───────────────────┼─┬─Query Restriction──────────┬Sort┐
   └────────────a─┤order_no           │n│                            │  │ │
                  │part_no            │y│                            │  │ │
                  │quantity           │y│                            │  │ │
                  │sale_price         │y│                            │  │ │
                  └───────────────────┴─┴────────────────────────────┴──┴─┘

  AddTable(SH-F1)  DelTable(SH-F2)  Edit(SH-F3)  AddJoin(SH-F4)  > :
```

The specifications shown in the figure correspond to the examples used in the discussion below.

Add New Records

The user of an Update frame can add data to the Master or Detail table (or both) in either of these ways:

- By updating a record, then using the AddNew menu operation to save the updated record as a new record. The original record is unaffected.
- By adding an entirely new record, then saving it with the AppendMode menu operation.

Vision generates menu items for both these operations when you allow users to add records on an Update frame.

On the visual query window, adding new records is controlled by a simple field labeled "Insert (y/n)" for the Master table and a similar field for the Detail table. By default, Vision lets users add new records to the Detail table but not to the Master table.

You can specify insertions separately for each table. For example, you can let users of an Update frame add new items to existing orders, but not add new orders.

Note: Although you can use Update frames to add new records, you cannot use any of the special features available on Append frames, such as sequenced fields and default values. (See the "Using Vision's Advanced Features" chapter for escape code that you can write to get around this restriction.)

To change the specifications for allowing users to add new records

1. Move the cursor to the Insert field for the Master or Detail table whose setting you are changing.
2. Type n or y as appropriate to change the setting.
3. If desired, repeat for the other table in the visual query.

Delete Records

The process of updating records on an Update frame can involve deleting records, also. By default, Vision lets users delete records from the Master and Detail tables. The following table describes the visual query specifications that control the deletion of records on an Update frame:

Specification	Default
Delete field for the Master table	Allow deletions in the Master table
Delete field for the Detail table	Allow deletions in the Detail table
Delete Cascades frame behavior	Delete all corresponding records in the Detail table when the user deletes a record in the Master table

Specifying Frame Behaviors (see page 145) describes how to set the Delete Cascades frame behavior.

If an Update frame has only a Master table, you can control deletions by setting the value of the Delete field for the Master table, as described in the procedures at the end of this section.

For an Update frame with a Master table and a Detail table there are two considerations involved in allowing Detail table row deletions:

- Whether to allow users to delete individual Detail table rows

  You control this specification through the Detail table Delete field on the visual query display, using the procedures described at the end of this section. When you allow Detail table row deletions, Vision generates a RowDelete menu item on the Update frame.

- When a user deletes a Master table record, whether to delete all corresponding Detail table rows

  You control this specification through a combination of the Master table Delete field on the visual query window and the Delete Cascades frame behavior, as described in the following table:

Deletions Desired	Setting for "Master Table Delete" Field	Setting for "Delete Cascades" Frame Behavior
None	N	N/A
Detail table only	N	N/A
Master table record and corresponding Detail table records	Y (default)	Cascaded (default)
Master table deletion allowed only if no corresponding Detail table records	Y (default)	Restrict
Master table only	Y	DBMS (The DBMS decides whether to delete corresponding Detail table rows, possibly resulting in "orphaned" rows in the Detail table.)

In the previous table, "N/A" indicates that the value for this specification does not affect the user actions allowed. In these cases, you must accept the default value.

Using the order entry example, you could let users delete order items from the Detail table, but not delete entire orders from the Master table. In this case, you would specify Detail table deletes only.

You could have another Update frame that displays open orders. You could specify Master table deletions allowed and Restrict Cascades frame behavior. Users now can delete orders from the Master table only if there are no open items in the Detail table.

To change the specifications for deleting records

1. Move the cursor to the Delete field for the Master or Detail table whose setting you want to change.
2. Type n or y as appropriate to change the current setting.
3. If desired, repeat for the other table in the visual query.

If necessary, change the Delete Cascades frame behavior as described in Specifying Frame Behaviors (see page 145).

Specify Query Restrictions

You can restrict the records that users can retrieve on an Update frame. You do this by specifying a query restriction in the visual query.

You can specify query restrictions on as many columns as you like in both the Master and Detail tables. These restrictions can be in addition to any query qualifications that the user enters when running the frame.

For example (see the preceding figure), you could specify that users only can retrieve orders whose total is less than $500. To do this, enter "<500" as a query restriction for the "order_total" column in the Orders table.

To specify a query restriction for a column

1. Move the cursor to the name of the column.
2. Tab to the Query Restriction field.
3. Enter a restriction for the column.

   The value of the restriction must be consistent with the column's data type. The value can be in the form of an expression as described in Using Expressions in Frame Definitions (see page 155).
4. If desired, repeat this procedure for additional columns.

Specify the Column Sort Order

By default, Vision retrieves records in whatever order they appear in the tables in the database. You can use the Visual Query Editor to indicate a specific order in which to sort retrieved records.

You can sort retrieved records:

- On any number of columns
- In a specified sequence of columns, regardless of the sequence of the columns in the table
- On both displayed and non-displayed columns

For example, you can display retrieved records in order by dollar amount first and then by customer number, even though a different column appears first in the table.

To specify the sort order of columns in the Master or Detail table

1. Move the cursor to the name of the Master or Detail table column.
2. Tab to the first column under the Sort label.
3. Type the number 1.
4. Press Tab again.

   The cursor moves to the second column under the Sort label.
5. Specify ascending (smallest to largest or "a" to "z") or descending order:
   - To sort the column in ascending order, press Tab. Vision enters an "a" as the default.
   - To specify descending sort order, type d.
6. Repeat the above sequence for each column on which you want to sort, entering the appropriate number and ascending/descending indicator.

To change the sort sequence after you specify it, move the cursor to the row for the column you want to change and enter a new sequence number. Vision adjusts the sort number for each column that comes later in the sequence.

How You Can Control Window Clearing

By default, Update and Browse frames on which the qualification processing frame behavior is enabled (see Specifying Frame Behaviors (see page 145)) are cleared each time a user selects Go. You can prevent the window from being cleared by setting the internal 4GL variable "IIclear2" to "n" in escape code.

For example, put the following statement in a Form-Start escape:

```
IIclear2 = 'n';
```

For instructions on how to write escape code, see Using Vision Advanced Features (see page 177).

How You Can Specify Unique Keys

The tables that you use with Update frames must have unique keys defined. Otherwise, when a user attempts to update or delete a record on the Update frame, Vision considers each column in the table to be part of a "unique key." This can lead to two types of problems:

- Performance is slower, because the query is constructed with all the columns as part of the where clause.
- A table with many columns can cause buffer overflow errors at run time, because of the size of the where clause.

To avoid these problems, use the SQL modify statement or create unique index statement to define a *unique* key on a table. (See the *SQL Reference Guide* for more information on these statements.) You then can use the Vision Reconcile utility so that the visual query reflects the table's new unique key. For more information on the Vision Reconcile utility, see Reconciling Tables and Frame Definitions (see page 251).

How You Can Specify Nullable Keys

When defining a table to use as a Master or Detail table on an Update frame, avoid creating any of the key columns as nullable. Nullable key columns cause Vision to generate additional source code, including a more complicated where clause, additional hidden fields, and an if-endif block before the update or delete statement.

This additional code prevents the generated where clause from failing when a user enters a null value into a key column. However, the additional code also makes the source code file larger and more difficult to read, and decreases the efficiency of the frame.

Defining Browse Frames

Browse frames let the user retrieve records from tables in the database. The user cannot perform any operations on these records.

Define a Browse frame to control the ways in which the user can retrieve records by using the Visual Query Editor to specify the following items:

Column Sort Order

You can specify the order in which to sort the records in the Master table and Detail table.

Query Restrictions

You can enter qualifications to restrict the records that the frame displays to the user.

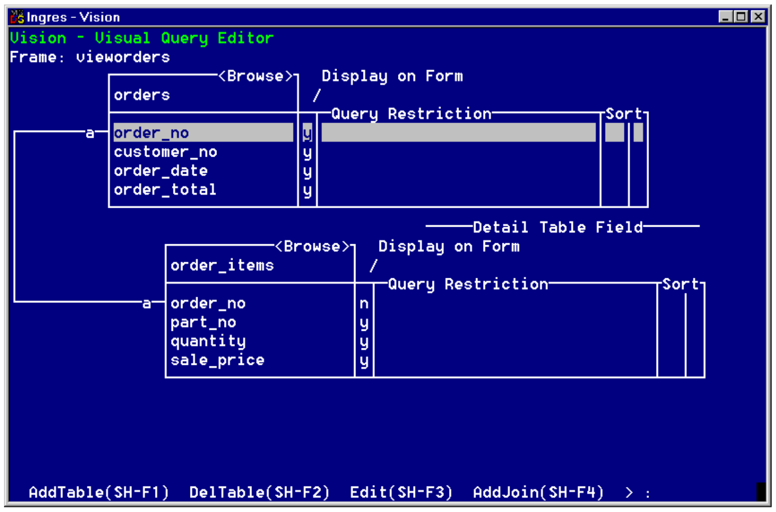
Qualification Processing

You can specify whether to allow users to enter query restrictions on the records that the frame retrieves.

Next Master Menu item

You can specify whether to display only the first Master table record (and any corresponding Detail table records) that satisfies the frame's query, or to have Vision generate a Next menu item that lets the user retrieve multiple Master table records.

The following figure shows the visual query window for a Browse frame:



```
Vision - Visual Query Editor
Frame: vieworders
                        <Browse>   Display on Form
          orders                    /
                                    Query Restriction              Sort
     a    order_no             y
          customer_no          y
          order_date           y
          order_total          y

                                           Detail Table Field
                              <Browse>   Display on Form
                order_items               /
                                          Query Restriction              Sort
           a    order_no             n
                part_no              y
                quantity             y
                sale_price           y

  AddTable(SH-F1)  DelTable(SH-F2)  Edit(SH-F3)  AddJoin(SH-F4)  > :
```

You specify the column sort order and query restrictions and control for Browse frames as described in Defining Update Frames (see page 137). Qualification Processing and Next Master Menuitem specifications are discussed in the next section.

For information about controlling window clearing on Browse frames that allow user qualifications, see Defining Update Frames (see page 137).

Specifying Frame Behaviors

When Vision generates the code for an Update or Browse frame, it specifies default behaviors for the frame. These behaviors control certain operations that users of the frame can perform. In most cases these default behaviors are appropriate; you rarely need to change them.

Summary of Frame Behavior Options

The following table provides a summary of the available frame behavior specifications for Browse and Update frames.

Behavior	Frame Types	Default	Alternatives
Qualification Processing	Browse, Update	Enable (allow user query qualifications)	Disable (prohibit user query qualifications)
Next Master Menuitem	Browse, Update	Enable (include Next menu item for multiple Master table row retrieval)	Disable (no Next menu item; allow single Master table row retrieval only)
Locks Held on Displayed Data	Update	None (displayed data is not locked from other users while being updated)	DBMS (displayed data is locked from other users until the update is committed) Optimistic (data is updated only if the row has not been changed by another user since the row was retrieved from the database)
Update Integrity Rule	Update	Cascade (updates to the Master table join column also cause updates of corresponding Detail table rows)	Restrict (updates to the Master table join column not allowed if corresponding Detail rows exist); or DBMS (use DBMS rule to handle Detail table join column updates)
Delete Integrity Rule	Update	Cascade (Master table deletions cause deletion of corresponding Detail	Restrict (Master table deletions not allowed if corresponding Detail table rows exist); or DBMS (use DBMS rule to handle Detail table

Behavior	**Frame Types**	**Default**	**Alternatives**
		table rows)	deletions)

The following sections describe each of the defaults and the alternative behaviors in detail. The last section provides steps you can use to change the defaults, if necessary.

Note: You can combine the various frame behaviors to control how users retrieve records on a Browse or Update frame.

Qualification Processing

By default, when the user calls a Browse or Update frame, the displayed form is blank. The user enters a query qualification, then selects Go to retrieve selected records from the database. For example, if your application uses a large table of customer information, users can enter qualifications to locate a particular customer record quickly.

User qualifications can be in addition to or instead of query restrictions that you specify when you define the frame. Users cannot enter query qualifications on Detail tables; they can enter qualifications for Master tables and Lookup tables only.

Change the user query qualification default so that the Browse or Update frame displays the first appropriate record from the Master table and Detail table, if any. This record can be any of the following:

- The first record that satisfies any query restrictions that you specify in the frame definition
- A specific record containing values passed as parameters from the parent frame (for details on passing parameters, see Passing Parameters Between Frames (see page 177))

  For example, a user can view a customer record on a Browse frame (the parent frame), then call an Update frame to display the same record to update.
- The first record stored in the database, if you do not use either of the above methods

Disable qualification processing when you want to pass parameters from a parent frame to a child frame. The parameters passed become the query qualifications for the child. In this way, the application controls which record the user sees; when qualification processing is disabled, the user cannot enter a query qualification to retrieve any other record.

If you disable qualification processing, you cannot use a Lookup table to provide a selection list, because the user cannot select the data to be displayed. However, you can use a Lookup table to enter data values into fields on the form. See Including Lookup Tables in Visual Queries (see page 123) for details.

Next Master Menu Item

By default, Vision generates a Next menu item for an Update or Browse frame. The user uses this operation to keep selecting any or all of the records in the Master and Detail tables (or the records that meet any query qualifications that you or the user has specified).

Change this default so that the Next menu item does not appear. In this case, Vision retrieves only the first Master table record or the first record that meets the query qualifications; the user cannot retrieve any additional records.

Do not use a Next menu item on a frame where the user needs to view or update one specific record only; for example, information about a particular customer.

If you have specified that the Master table appear on the form as a table field, you cannot use the Next Master Menu Item frame behavior.

Locks Held on Displayed Data

By default, Vision does not lock the data that a user is displaying on an Update frame. This means that multiple users are allowed to browse and update the same data at the same time. The risk, of course, is that one user can overwrite data just entered by another user.

You can change the default so that Vision holds a lock on the displayed data. The frame behavior Locks Held on Displayed Data controls locking. There are three choices for this frame behavior:

- None
- DBMS
- Optimistic

This frame behavior applies only to Update frames.

DBMS or Shared Locking

DBMS, or shared, locking means that other users are blocked from changing the selected data until the first user selects Save or otherwise releases the lock. The other users can select the same displayed data, but cannot update it until the first user has saved any changes.

If you design your application so that Vision holds shared locks, the displayed data is locked when the user selects Go. The locks on the Master table data are dropped when a user:

- First selects Save
- Moves past the last selected Master table row by selecting Next
- Selects End to leave the query or the frame
- Selects a menu item that calls a browse frame. The select loop of a browse frame issues a commit that releases the locks, including the locks on the calling frame's data.

Be aware that shared locks can result in deadlock when multiple users hold a shared lock on the same data and each selects Save. For information on how Vision handles deadlocks, see How Vision Handles Deadlocks (see page 338).

Optimistic Locking

If you use optimistic locking, Vision holds no locks while viewing data. When a user (user1) tries to save a record, Vision checks whether the record has been retrieved and updated by another user (user2) since user1 retrieved it from the database. If the record has been updated by user2, user1's changes are not saved. If the record has not been updated, user1's changes are saved. Optimistic locking depends on Vision using columns in your database tables to track row updates.

To use optimistic locking, you must specify the columns for Vision to use for tracking when the row was last updated. Each column can contain one of the following types of information:

- A date/time stamp, showing when each row of the table was last updated
- An update counter, or sequence value, which is incremented each time the row is updated
- The name of the user who last updated each row

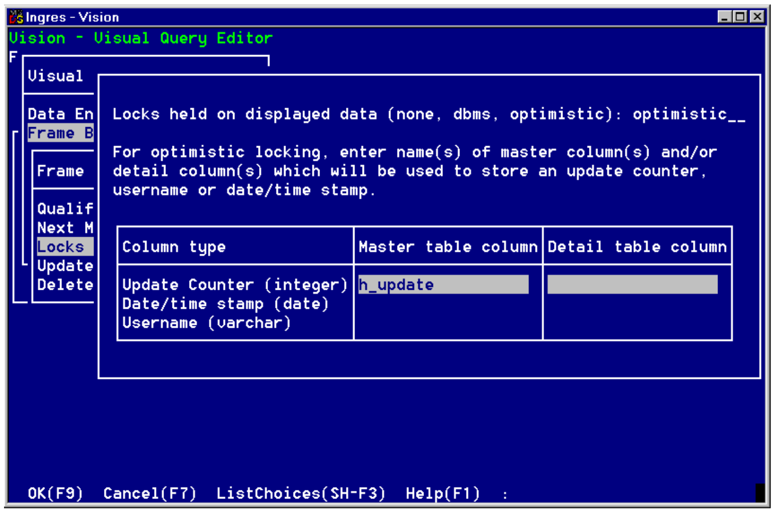
To use one or more of these types, the recommended combinations are:

- Date/time stamp column alone
- Date/time stamp column, with the user name. This adds an additional check in case more than one user is updating the data at the same time
- Update counter column alone. An update counter is sufficient, but you can also add a username if you would like this information for tracking purposes

Note: A user name by itself does not provide a sufficient check.

These columns can be displayed on your form or they can be hidden columns. They must exist in your database tables.

Specify the columns on the Locks Held on Displayed Data pop-up, as shown in the following figure. (See the directions in Change a Frame Behavior Specification (see page 151) to get to the pop-up.)


Ingres - Vision
Vision - Visual Query Editor
F
Visual
Data En
Frame B
Frame
Qualif
Next M
Locks
Update
Delete
OK(F9) Cancel(F7) ListChoices(SH-F3) Help(F1) :

```
Locks held on displayed data (none, dbms, optimistic): optimistic__

For optimistic locking, enter name(s) of master column(s) and/or
detail column(s) which will be used to store an update counter,
username or date/time stamp.
```

Column type	Master table column	Detail table column
Update Counter (integer)	h_update	
Date/time stamp (date)		
Username (varchar)		

After you have specified the column names, Vision automatically uses the optimistic concurrency in the generated code. When the code is generated, the date/time stamp column is set to 'now', the user name column is set to user and the update counter column is set to 0. Every time a user updates the row, the date/time stamp column is set to 'now', the user name column is set to user and the update counter column is set to the previous value plus 1.

The Update Integrity Rule

If both the following conditions occur:

- An Update frame uses both a Master table and Detail table in its visual query
- A user changes the value in a join column for a record in the Master table

Then Vision's default action is to change that value for all *corresponding records* in the Detail table. A Detail table record corresponds to the current Master table record if it has the same value as the Master table in a join column.

Change this default to either of the following:

- Restrict updates so that the user cannot change the value of the join column in a Master table record if corresponding Detail table records exist
- Allow the DBMS to handle Detail join column updates based on a rule that has been created previously

A rule is a mechanism to invoke a database procedure whenever a specific condition is true. See the *SQL Reference Guide* for more information about rules and procedures.

This frame behavior only applies to Update frames.

The Delete Integrity Rule

The Delete Integrity Rule frame behavior affects the deletion of records from Update frames with both Master and Detail tables. The default behavior is to delete any corresponding Detail table records when a user deletes a Master table record.

Use the Delete Integrity Rule frame behavior in combination with the Delete specifications on the visual query window. See Defining Update Frames (see page 137) for a discussion of how to control the deletion of records on Update frames.

You also can specify that Vision use a DBMS rule that you have created previously to determine how to handle deletions to Detail table records. A rule is a mechanism to invoke a database procedure whenever a specific condition is true. See the *SQL Reference Guide* for more information about rules and procedures.

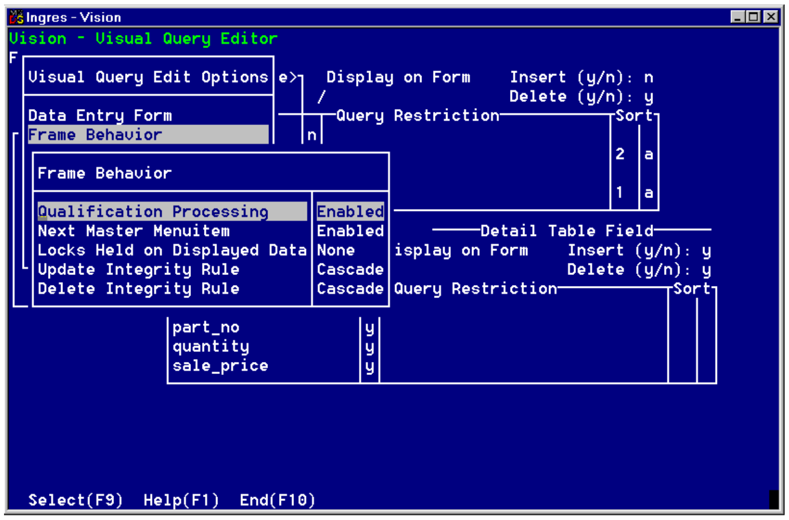
Change a Frame Behavior Specification

To change any of the frame behavior specifications

1. Select Edit from the Visual Query Editor menu.
2. Select Frame Behavior from the list of edit options.

   Vision displays the list of frame behaviors.

   The following figure shows the current specification for each behavior:



```
Ingres - Vision
Vision - Visual Query Editor
F
 Visual Query Edit Options e>   Display on Form    Insert (y/n): n
                              /                    Delete (y/n): y
 Data Entry Form              Query Restriction                Sort
 Frame Behavior             n
  Frame Behavior                                           2   a

                                                           1   a
  Qualification Processing     Enabled
  Next Master Menuitem         Enabled      Detail Table Field
  Locks Held on Displayed Data None    isplay on Form     Insert (y/n): y
  Update Integrity Rule        Cascade                    Delete (y/n): y
  Delete Integrity Rule        Cascade Query Restriction              Sort

                 part_no              y
                 quantity             y
                 sale_price           y



 Select(F9)  Help(F1)  End(F10)
```

3. Select the frame behavior you want to change.

   Vision displays the specification window for the behavior you have selected.

4. If you have selected Locks Held on Displayed Data, Vision displays the window shown in the figure in Optimistic Locking (see page 147).

   a. Enter the desired setting (none, optimistic, or dbms) for the behavior in the Locks held on displayed data field.

      If you select the setting "Optimistic" for the Locks Held on Displayed Data frame behavior, you must also select the names of the columns used for locking. See Optimistic Locking (see page 147) for details.

   b. Choose OK from the menu.

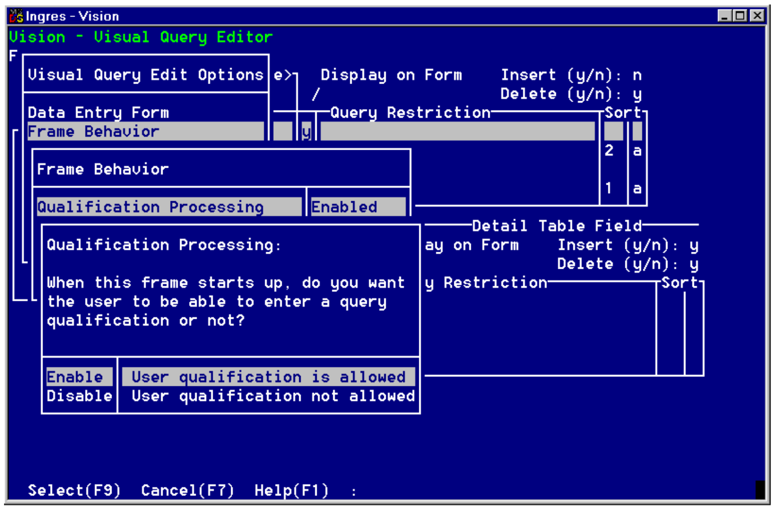
      For all other frame behaviors, Vision displays a window similar to the window for Qualification Processing shown in the following figure. The cursor is positioned on the current behavior.

   c. Move the cursor to the desired setting for the behavior.

   d. Choose Select from the menu.

   Vision changes the behavior specification and returns you to the frame behavior options list.

5. Choose another option or select End to return to the visual query display window.



```
Ingres - Vision
Vision - Visual Query Editor
F
 Visual Query Edit Options e>   Display on Form     Insert (y/n): n
                             /                      Delete (y/n): y
 Data Entry Form                  Query Restriction              Sort
 Frame Behavior                 y
                                                                  2  a
  Frame Behavior
                                                                  1  a
  Qualification Processing  Enabled
                                         Detail Table Field
   Qualification Processing:          ay on Form    Insert (y/n): y
                                                    Delete (y/n): y
   When this frame starts up, do you want  y Restriction       Sort
   the user to be able to enter a query
   qualification or not?

   Enable  User qualification is allowed
   Disable User qualification not allowed


 Select(F9)  Cancel(F7)  Help(F1)  :
```

Modifying Vision-generated Forms

Vision creates a default form for Append, Browse, and Update frames based on the current frame definition. Whenever you select End from the Visual Query Editor menu, Vision generates a new form for the frame, incorporating any changes you have made to the visual query.

However, if you use the forms editor to modify the form (as described in Editing a Form (see page 153)), Vision does not regenerate the form, but merges the existing form with the visual query changes to create a new form with the same name as the current one. This process is known as "form fixup."

In form fixup, Vision reflects the visual query changes on the new form by:

- Adding a simple field or table field column for any column whose display status you change to "yes"

  Vision places the new simple field at the top of the form; it places the new table field column at the front of the table field.
- Removing a field for any column whose display status you change to "no"
- Removing the table-field column for any database column in the Detail table for which you add a join
- Adding trim on the form to let you know what changes were made

Form fixup ensures that Vision does not overwrite any of the changes that you have made with the forms editor. However, you must edit the form again to incorporate the changes that Vision has made; for example, by moving the new fields to the appropriate location.

Editing a Form

Use Vision's forms editor to make additional changes to the form, including:

- Changing the position of a field or trim on the form
- Changing the order in which the user accesses fields
- Making a field on the form invisible to the user
- Specifying validation criteria for the data that the user enters into a field
- Requiring that the user enter a value for a particular field
- Changing the display attributes of a field (color, reverse video, etc.)
- Changing the title of a field
- Modifying the display format of a field; for example, changing the initial value of a sequenced field on an Append frame from "1" to "0001"
- Designating a form as full-window or pop-up
- Adding descriptive or explanatory lines of text
- Creating lines, boxes or other trim to make the form more attractive or easier to use
- Creating or deleting fields
- Deriving the value of a field from the values of other fields

Forms-based Querying and Reporting Tools User Guide provides detailed instructions for editing forms. Remember that even though a Menu frame does not have a visual query, Vision still generates a form for it that you can edit.

Call the Forms Editor

While you are defining a Vision frame, you can call the forms editor from several places to view or modify the associated form.

To edit a form from the Application Flow Diagram Editor

1. Select as the current frame the frame whose form you want to edit.
2. Select Edit from the menu.
3. Select Data Entry Form from the list of Frame Edit Options.

   Vision displays the form for this frame.

(As an alternative to Steps 2 and 3, select MoreInfo from the Application Flow Diagram Editor menu, then select FormEdit from the submenu.)

To edit a form from the Visual Query Editor

1. With the cursor positioned anywhere on the visual query display, select Edit from the menu.
2. Select Data Entry Form from the list of Visual Query Edit Options.

   Vision displays the form for this frame.

When you call the forms editor, Vision generates a new form for a frame if you have just created the frame or changed the frame definition.

If you have access to ABF, you also can call the forms editor through the Edit operation of the ABF Catalog window.

Coordinating the Form and the Visual Query

In order for Vision to generate the correct code and form for a frame, it is important that the form for a frame correspond to its visual query. Therefore, Vision places these restrictions on the changes you can make to a form:

- You cannot remove a field for a column in a table used in the visual query. You can, however, make the field invisible.

  To remove a field from the form, you must go back to the Visual Query Editor and change the display status of the column (that is, change the indicator in the Display on Form column to "n").

- You cannot change the data type or internal name of a field for a column in a table used in the visual query.

  The internal name is the name by which the column is stored in the database; this is the name shown in the visual query display.

- You must not create a new field with the same name as a column in a table used in the visual query. The forms editor lets you create the field, but an error occurs when you run the frame.

  (Be aware that this situation occurs if you replace the form with a duplicate form that you have created with VIFRED.)

- If you delete a field with an escape code activation, Vision deletes the escape code as well.

  For details on writing escape code, see Using Vision Advanced Features (see page 177).

Use a Form with Multiple Frames

You can use the same form with frames that have identical visual queries; for example, a Browse and an Update frame. In this way, you need to edit the form only once.

Note: You cannot share a form between two frames when one of the frames calls the other.

To specify a form name other than the default for a frame

1. Use the MoreInfo operation of the Application Flow Diagram Editor to enter the form name.
2. Use the Compile menu item of the Application Flow Diagram Editor to regenerate the code for the frame.

   This ensures that the source file refers to the correct form.

If you change the visual query for two Append, Browse, or Update frames that share a form, Vision updates the form for one frame—by removing a field, for example—in such a way that the form is incompatible with the other frame. This causes compile errors in the second frame.

This is not a problem if one of the frames is of a type for which Vision does not generate the code.

Using Expressions in Frame Definitions

When you are defining frames, you can specify a value in the form of an expression to:

- Assign values to columns on Append frames
- Specify query restrictions on Browse and Update frames
- Pass parameters between frames

  (The syntax for parameters can differ from the syntax described in this section. Parameters are discussed in Passing Parameters Between Frames (see page 177).)

The following table describes the types of expressions that you can use in Vision frame definitions.

Expression	**Examples**
A numeric or string constant	4 7.6 'NE'

Expression	Examples
	'today'
A simple field	:part_no
A table-field column	:detailtbl.name
An arithmetic expression using any of these operators: + - * / **	:salary*1.1 (110% of the value of the Salary field) :order_no+1 (1 greater than the value of the order number field)
A logical expression using any of these comparison operators: = ,!=,<>,^=,<,<=,>,>= AND,OR,IS NULL, IS NOT NULL,LIKE, NOT LIKE,IN,NOT IN	>2 * :total (greater than twice the value of the Total field) != 'Jones' (retrieves all values except 'Jones')
You can use logical expressions with Browse and Update frames only	IN (1,2,3) IN ('NW','SE')

Specifying Expressions

You include expressions in visual queries as "fragments" that Vision includes in the code it generates for a frame. Specify these code fragments according to the following rules:

- The data type of the expression must be compatible with the data type of the column.

  For example, do not use an integer value for an assignment or query restriction in a character column.

- Enclose character strings in single quotes.

- When assigning a value to a field on an Append frame or specifying a query restriction on a Browse or Update frame, do not include the name of the field, because Vision already knows the field name.

  For example, to restrict retrievals on a Browse frame to customer accounts with a balance of more than $1,000, enter the following as a restriction for the Balance column on the visual query window:

  ```
  >1000
  ```

  not:

  ```
  balance > 1000
  ```

- Use a colon (:) to indicate a reference to another field name.

  For example, to retrieve customer records on a Browse frame where the amount of the current order is greater than the account balance, specify the following as a query restriction for the order_total column:

  ```
  >:balance
  ```

  (You do not need to include a colon with a field name when you specify a parameter.)

- You can refer to either displayed or hidden fields in an expression.

- The "=" is optional in expressing equivalence.

  For example, to set the value of the Dept field to 'sales', you can enter either:

  ```
  'sales'
  ```

  or

  ```
  = 'sales'
  ```

- Use LIKE and NOT LIKE to retrieve character strings.

  For example, to retrieve employees whose last name begins with 'S,' enter the query restriction on the Browse or Update frame for the Lastname column as:

  ```
  LIKE 'S%'
  ```

  (The "%" matches any string of characters.)

- Use IN and NOT IN (or OR) to retrieve numeric values.

  For example, to retrieve only salaries equal to $10,000 or $50,000 on a Browse frame, enter:

  ```
  IN (10000,50000)
  ```

  or

  ```
  10000 OR 50000
  ```

  in the Query Restriction field for the Salary column on the visual query window.

For detailed information on using expressions and for more information about pattern matching and comparison operators, see other sections of this guide.

How You Can Use the dbmsinfo Function

You can use the dbmsinfo function to retrieve various types of information from the database, such as current user names or transaction states. You only can retrieve information into the non-displayed fields on a frame.

The dbmsinfo function has the format:

```
dbmsinfo ('request_name')
```

where *request_name* represents a specific item of information. See the *SQL Reference Guide* for all possible values of request_name.

You can use the dbmsinfo function as an expression on Append, Browse, and Update frames. For example, you could assign the following value to a non-displayed field on an Append frame:

```
dbmsinfo ('username')
```

When a user appends a record, Vision enters the user's name into the database column corresponding to the non-displayed field.

Chapter 7: Defining Frames without Visual Queries

This section contains the following topics:

Vision generates Menu, Append, Browse, and Update frames that let you include a wide range of user operations and query specifications in your applications. In addition, you can increase the scope of your applications by including frames that access other Ingres tools.

For example, you can create a frame to display a report created with RBF or to run a query defined in QBF.

You also can include frames and procedures for which you write the code.

Because Vision does not generate the code for these frame types, they do not have visual queries. To use one of these frame types in an application you must:

- Create a new frame in the Application Flow Diagram (as described in Creating Frames (see page 87)) specifying the appropriate frame type
- After you create the frame, define it to Vision; that is, provide information about its components:
  - Forms
  - Query object (such as a report or QBF JoinDef) or source code file

  These components can exist before you create the frame or, in most cases, you can create them at the same time. All the components must reside in the same database as the Vision application in which you use them.

This chapter describes how to define frames and procedures that do not have visual queries. These include:

- QBF frames
- Report frames
- Graph frames
- User frames

- 4GL procedures
- 3GL procedures
- Database procedures

For a description of each of these frame and procedure types, see Overview of Vision (see page 59).

You cannot create child frames in the Application Flow Diagram for any of the frame types listed above. However, you can call other frames through the 4GL code for User frames and procedures (with certain exceptions, as described in this chapter). Frames called in this way do not appear in the Application Flow Diagram.

For information on how to use ABF to create frames and call them from your application, see Using Vision or ABF in a Multi-Developer Environment (see page 1381).

Frame Definition Menu Operations

Vision provides various menu operations to use when defining a frame or procedure. These operations also are available if you later edit the frame definition.

The following operations appear on the MoreInfo about a Frame or Procedure windows. The MoreInfo window for each specific frame or procedure type is described in the appropriate section of this chapter.

Rename

Lets you change the name of a frame or procedure

LongRemark

Displays a pop-up window in which you can enter a comprehensive description of the frame or procedure

ListChoices

Provides a list of valid responses for various fields on the MoreInfo window

Cancel

Returns you to the Application Flow Diagram Editor without saving any changes you have made to the MoreInfo window

Help

Provides information to help you define the specific frame or procedure type

End

Saves your frame definition and returns you to the Application Flow Diagram Editor

Define a QBF Frame

You can include frames in your Vision applications to run database queries that you have defined previously through QBF. Creating new QBF queries is not necessary because Append, Browse and Update frames let you define your queries more conveniently and efficiently.

See *Forms-based Querying and Reporting Tools User Guide* for more information about specifying queries with QBF.

To define a QBF frame

1. Follow Steps 1 through 8 in Create a New Frame (see page 98). Specify "QBF" as the frame type.

   Vision redisplays the Application Flow Diagram with the new frame inserted.

2. Select MoreInfo to display the MoreInfo About a QBF Frame Window, as shown in the following figure. The window displays the frame name and description you just entered, as well as when and by whom the frame was created.

3. Specify the type of query object for the frame in either of these ways:

   - If the frame uses a QBF Join Definition, accept the default value of JoinDef by pressing Tab.
   - If the frame uses a table, type table and press Tab.

4. Enter the name of the JoinDef or table and press Tab.

   The table or QBF Join Definition must exist in the database already.

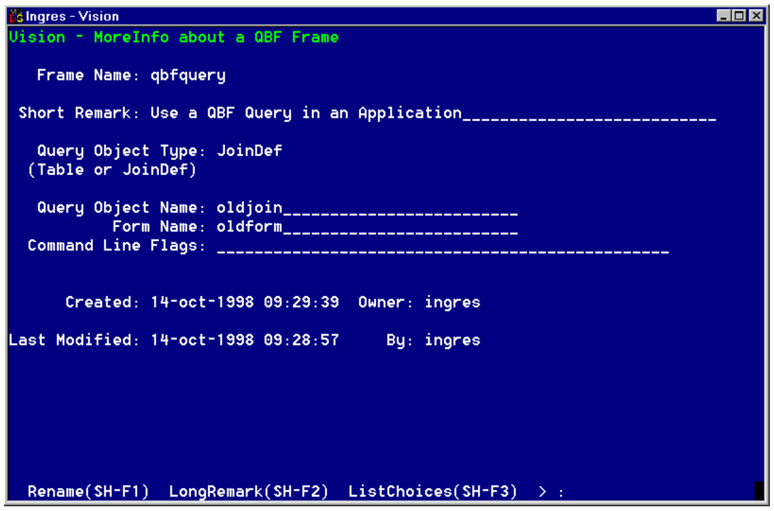
5. Enter the name of the form associated with this frame and press Tab.

6. Enter any command line flags—that is, any parameters of the query command to use when this frame is called—and press Tab.

7. If needed, you can edit the descriptive Short Remark that you entered when you created the frame.

   When you are done, select End to return to the Application Flow Diagram.

The following figure shows the MoreInfo about a QBF frame window:



```
Vision - MoreInfo about a QBF Frame

   Frame Name: qbfquery

 Short Remark: Use a QBF Query in an Application___________________________

   Query Object Type: JoinDef
  (Table or JoinDef)

   Query Object Name: oldjoin_________________________
           Form Name: oldform_________________________
  Command Line Flags: ________________________________________________


      Created: 14-oct-1998 09:29:39  Owner: ingres

Last Modified: 14-oct-1998 09:28:57     By: ingres




  Rename(SH-F1)  LongRemark(SH-F2)  ListChoices(SH-F3)  > :
```

View and Edit a QBF Definition

While you are working in Vision you can access the query object of a QBF frame. Specifically, you can:

- Call the Tables Utility to examine a table
- Call QBF to modify a JoinDef

To access the query object of a QBF frame

1. Select the QBF frame as the current frame in the Application Flow Diagram.
2. Select Edit from the menu.
3. Select QBF Definition from the Frame Edit Options list.

   Vision calls the Tables utility for table query objects or the QBF editing window for JoinDef query objects.
4. Change any of the query specifications as desired. Be sure to save your changes within QBF or the Tables utility.
5. Select End to return to the Application Flow Diagram.

Define a Report Frame

Your Vision applications can include frames that call reports from the database. You can create these reports with either RBF or the Report-Writer report specification language. The report can already exist, or you can create it when you create the Report frame.

See *Character-based Querying and Reporting Tools User Guide* for more information about creating and running reports.

To define a Report frame

1. Follow Steps 1 through 8 in Create a New Frame (see page 98). Specify "Report" as the frame type.

   Vision redisplays the Application Flow Diagram with the new frame inserted.

2. Select MoreInfo to display the MoreInfo About a Report Frame Window, as shown in the following figure. The window displays the frame name and description you just entered and when and by whom the frame was created.

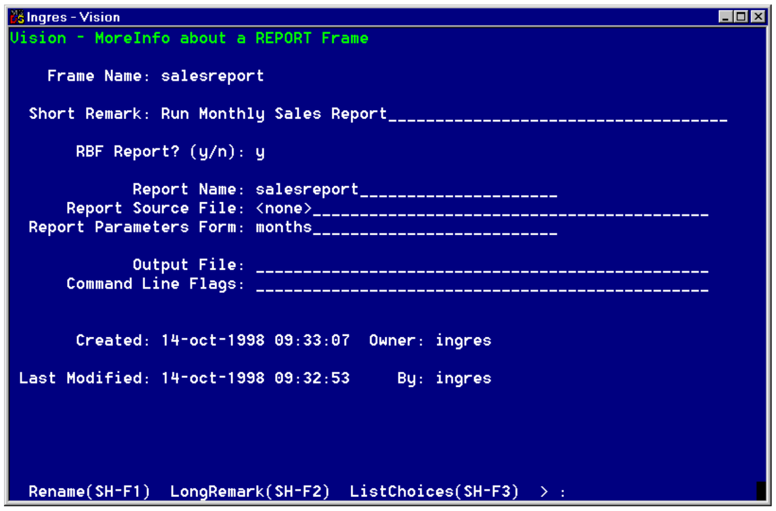
3. Specify the report type in the RBF Report? field:

   - Press Tab to accept the default of a report created in RBF.
   - For Report-Writer reports, type n.

   The cursor moves to the Report Name field. Vision assumes that the report name is the same as the frame name.

4. Press Tab to accept this default report name or enter a new name and press Tab.

   The report need not exist yet in the database. Before you run the frame, you can create the report as described in the Creating and Editing a Report Definition section. However, it is more efficient to create the report outside of Vision before you create the Report frame.



```
Ingres - Vision
Vision - MoreInfo about a REPORT Frame

   Frame Name: salesreport

 Short Remark: Run Monthly Sales Report____________________________________

      RBF Report? (y/n): y

            Report Name: salesreport_____________________
     Report Source File: <none>________________________________________
 Report Parameters Form: months__________________________

            Output File: ________________________________________________
     Command Line Flags: ________________________________________________


      Created: 14-oct-1998 09:33:07  Owner: ingres

Last Modified: 14-oct-1998 09:32:53     By: ingres



 Rename(SH-F1)  LongRemark(SH-F2)  ListChoices(SH-F3)  > :
```

5. If you have specified a report created with the Report-Writer, enter the name of the file containing the report specifications, then press Tab.

   If you specified RBF as the report type, the Report Source File name appears as "none."

6. Enter in the Report Parameters Form field the name of a form on which the user can specify runtime parameters for the report (see Specifying Runtime Report Parameters (see page 165)).

7. If you want to print the report to a file, enter the file name in the Output File field.

   Leave the field blank to print the report to your window.

   For a Report-Writer report, a file you specify here overrides any file specified in an .output statement. Be sure to include the complete directory path if you do not want the file you specify to be written to the current directory.

8. Specify in the Command Line Flags field any parameters of the report command to use when this frame is called.

9. Edit the descriptive Short Remark that you entered when you created the frame.

10. When you are done, select End to return to the Application Flow Diagram.

Specifying Runtime Report Parameters

You can let users of your application restrict the data that a Report frame displays. For example, if your Report frame runs a report containing three years' worth of sales results, when a user calls the frame, the user can ask to see the results for the past six months only.

You can specify such runtime data selection criteria in either of two ways:

- By passing the criteria as parameters from the Vision frame that calls the Report frame (see Passing Parameters Between Frames (see page 177))

  For example, if you have a Browse frame to display information about sales results, this frame has a Region field whose value indicates a geographic area.

  You can pass the current value of Region on the Browse frame to a Report frame. When the user calls the Report frame from the Browse frame, a report is produced with data for that region.

- By letting the user respond to prompts that you include when you specify the report (with either RBF or the Report-Writer)

  You can create a form that contains a field for each report parameter. When the user calls the Report frame, Vision displays this form, and the user can enter the values to use to run the report.

  You create this form with the VIFRED outside of Vision before you run the Report frame. You specify the name of this form on the MoreInfo about a Report Frame window.

  If you do not create a form for the frame, Vision displays your prompts on a blank window when the user calls the Report frame.

Create and Edit a Report Definition

You can call RBF or a Report-Writer specification file directly from Vision to create or edit a report. You can create the report at any time before you run the frame.

To create or edit a report for a Report frame

1. Select the frame as the current frame in the Application Flow Diagram.
2. Select Edit from the menu.
3. Select Report Definition from the Frame Edit Options list.

   Vision calls the appropriate report specification Tool.

4. Change any of the report specifications as desired. Be sure to save your changes within RBF or the Report-Writer file.
5. Select End from RBF or close the Report-Writer specification file to return to the Application Flow Diagram.

Define a User Frame

You can include User frames in your Vision applications. You must write the full 4GL specifications (including the necessary menu items for any frames that your User frame calls) and create a form in VIFRED for a User frame. You can include User frames that you previously created, or use Vision to create new User frames.

See the 4GL reference part of this guide for more information about coding User frames. See *Character-based Querying and Reporting Tools User Guide* for more information about creating forms for User frames.

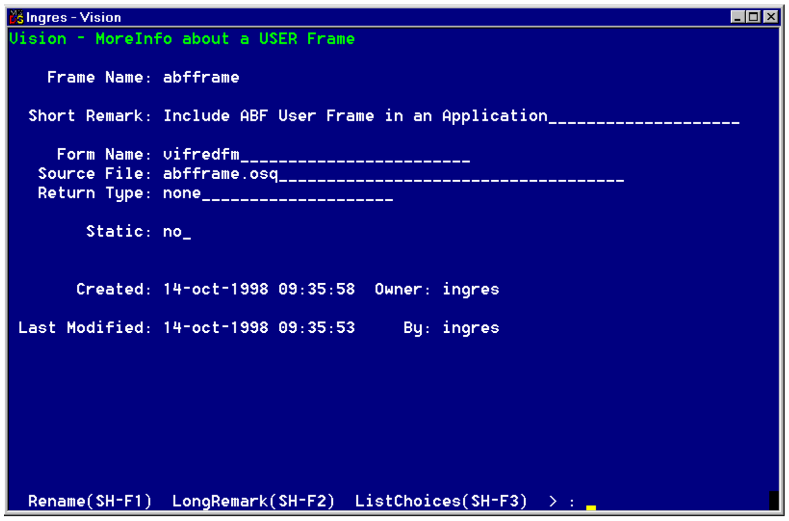
To define a User frame

1. Follow Steps 1 through 8 in Create a New Frame (see page 98). Specify "User" as the frame type.

   Vision redisplays the Application Flow Diagram with the new frame inserted.

2. Select MoreInfo to display the MoreInfo About a User Frame window, as shown in the following figure. The window displays the frame name and description you just entered, as well as when and by whom the frame was created.

   When the window is displayed, the cursor is on the Form Name field. Vision assumes that the form name is the same as the frame name.



```
Vision - MoreInfo about a USER Frame

    Frame Name: abfframe

  Short Remark: Include ABF User Frame in an Application___________________

     Form Name: vifredfm________________________
   Source File: abfframe.osq____________________________________
   Return Type: none____________________

        Static: no_


       Created: 14-oct-1998 09:35:58  Owner: ingres

 Last Modified: 14-oct-1998 09:35:53     By: ingres



  Rename(SH-F1)  LongRemark(SH-F2)  ListChoices(SH-F3)  > :
```

3. Press Tab to accept this default form name, or enter a new name and press Tab.

   The form for this frame need not exist yet. You can create it later, as described in the Creating and Editing the 4GL Code and Form section.

   The cursor next moves to the Source File field. Vision assumes that the Source File, which contains the 4GL code for this frame, has the same name as the frame and a ".osq" extension.

4. Press Tab to accept this default file name or enter a new name and press Tab.

   The source code for this frame need not exist yet. You can create it later, as described in the Creating and Editing the 4GL Code and Form section.

   The cursor moves to the Return Type field. The Return Type indicates the data type of a value passed from this frame back to the parent frame; the default is "none."

5. Press Tab to accept the default return type or enter a new return type.

   You can specify the return type as any valid Ingres data type or use a record definition that you have created for this application.

   The cursor moves to the Static field. The default value of "no" indicates that Vision reinitializes any local variables each time it calls this frame, rather than using the current values.

6. Press Tab to accept the "no" default value or type yes and press Tab.

   If you enter yes, Vision uses the current values of any local variables for this frame, rather than reinitializing them when it calls the frame.

7. If needed, you can change the descriptive Short Remark that you entered when you created the frame.

8. When finished, select End to return to the Application Flow Diagram.

Create and Edit the 4GL Code and Form

You can create or edit the 4GL source code file for a User frame directly from Vision. You also can call VIFRED to create or edit the form for a User frame.

You can create the source code file and form at any time before you run the frame.

To create or edit the 4GL code or form for a User frame

1. Select the frame as the current frame in the Application Flow Diagram.
2. Select Edit from the menu.

3. Select the appropriate frame component:
   - Data Entry Form to edit the form
   - Source Code File to open the source code file.

     Use VIFRED to edit the form or the system editor to edit the 4GL source code. Be sure to save any changes that you make.

     If you are working with a new User frame, Vision lets you create the new frame component based on your selection as follows:
   - If you selected Data Entry Form, Vision calls VIFRED and displays the Creating a Form window. See *Character-based Querying and Reporting Tools User Guide* for more information about creating forms in VIFRED.
   - If you selected Source Code File, Vision calls the system editor and opens a blank file in which you can write the 4GL source code for the frame.
4. To return to the Application Flow Diagram:
   - Select End from VIFRED.
   - Exit from the source code file.

Controlling Activations and Validations

The Vision code generator issues a set_forms frs statement for generated frames (Menu, Append, Browse, or Update) that sets activations and validations to "1." These settings apply to all frames in your Vision application, including any User frames that you have defined.

If you do not want your User frame to use these settings, override the generated settings, by including a set_forms frs statement in the code for the User frame that explicitly sets activations and validations to "0."

Including Procedures in an Application

In addition to frames your Vision applications can include *procedures;* that is, series of command statements. When your application calls a procedure, Vision executes the procedure's commands.

Write a procedure with statements written in:

- 4GL
- A 3GL, sometimes called a host language
- SQL

See the 4GL reference part of this guide, the *SQL Reference Guide,* or the *Companion Guide* for your particular 3GL for more information about creating procedures.

Follow these guidelines when using procedures in your Vision applications:

- 4GL procedures cannot contain references to forms or fields on a form.
- 3GL procedures cannot call 4GL frames or procedures.
- Procedures can call other procedures.
- You can use 4GL and 3GL procedures that already exist, or you can write the code when you include them.
- Before you can include an SQL database procedure in a Vision application, you must have written the procedure previously with the SQL create procedure statement

  See the *SQL Reference Guide* for more information about creating database procedures.

You create and define procedures for your applications just as you do frames. Each section below describes how to define a procedure type.

Define a 4GL Procedure

To define a 4GL procedure

1. Follow Steps 1 through 8 in Create a New Frame (see page 98). Specify "4GL Proc" as the type.

   Vision redisplays the Application Flow Diagram with the new procedure inserted.

2. Select MoreInfo to display the MoreInfo About a 4GL Procedure window as shown in the following figure. The window displays the procedure name and description you just entered, as well as when and by whom the procedure was created.

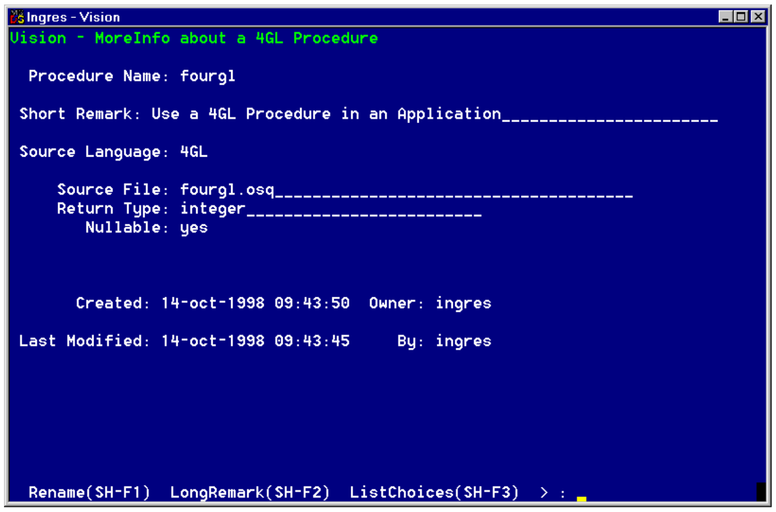
   When the window is displayed, the cursor is on the Source File field. Vision assumes that the 4GL source code file has the same name as the procedure and the ".osq" extension.

3. Press Tab to accept this default file name, or enter a new file name and press Tab.

   The cursor next moves to the Return Type field. This value indicates the data type of a value passed from this procedure back to the calling frame or procedure. The default value is "integer."

4. Press Tab to accept the default value or enter any of the following:
   - None to indicate no return type
   - A standard Ingres data type
   - A record type that you have created for this application. For information about creating record types, see Using Record Types (see page 205).

     The cursor next moves to the Nullable field. The default value of "yes" indicates that any return data can have a null value.
5. Press Tab to accept the default value, or type no and press Tab to prohibit a null value for return data.



```
Vision - MoreInfo about a 4GL Procedure

  Procedure Name: fourgl

 Short Remark: Use a 4GL Procedure in an Application_______________________

 Source Language: 4GL

     Source File: fourgl.osq______________________________________
     Return Type: integer_________________________
        Nullable: yes



       Created: 14-oct-1998 09:43:50  Owner: ingres

 Last Modified: 14-oct-1998 09:43:45     By: ingres





  Rename(SH-F1)  LongRemark(SH-F2)  ListChoices(SH-F3)  > :
```

6. If needed, you can edit the descriptive Short Remark that you entered when you created the procedure.
7. When you are done, select End to return to the Application Flow Diagram.

Define a 3GL Procedure

You can include in your Vision applications procedures written in a 3GL. A 3GL is a programming language such as C, which can contain embedded database and forms statements. You must use a language for which you have purchased a Ingres preprocessor.

If you use a language for which you do not have an Ingres preprocessor, do not include any database access or forms display statements. You can use such "unpreprocessed" procedures to perform calculations on data and return a value.

The 3GL procedure can exist in a source file or as a library routine. If the procedure is a library procedure, you must specify the name of the link options file, as described below.

To define a 3GL procedure

1. Follow Steps 1 through 8 in Create a New Frame (see page 98). Specify "3GL Proc" as the type.
2. Specify the language in either of these ways:

   Choose a language from the selection list.

   - Vision lists those languages for which you have purchased a preprocessor.
   - Select OtherLanguages from the menu and select the name of another programming language.

   Vision redisplays the Application Flow Diagram with the new procedure inserted.
3. Select MoreInfo to display the MoreInfo About a 3GL Procedure Window as shown in the following figure. The window displays the procedure name and description you just entered, as well as when and by whom the procedure was created.

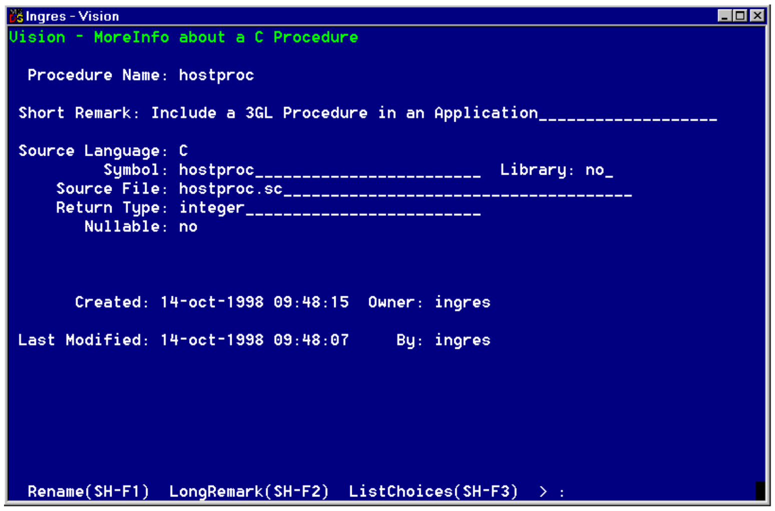
   When the window is displayed, the cursor is on the Library field. The default value of "no" indicates that the code for the procedure is contained in a source code file.

4. Press Tab to accept the default value or type yes to indicate that the procedure exists as a library routine.

   If you specified "yes," proceed to Step 5.

   If you specified "no," the cursor moves to the Source File field. The field contains a default file name based on the procedure name and the appropriate file extension for the language you specified.

   You cannot specify a source code file for a library procedure.



```
Ingres - Vision
Vision - MoreInfo about a C Procedure

  Procedure Name: hostproc

 Short Remark: Include a 3GL Procedure in an Application___________________

 Source Language: C
          Symbol: hostproc________________________  Library: no_
     Source File: hostproc.sc_____________________________________
     Return Type: integer_________________________
        Nullable: no



       Created: 14-oct-1998 09:48:15  Owner: ingres

 Last Modified: 14-oct-1998 09:48:07     By: ingres





  Rename(SH-F1)  LongRemark(SH-F2)  ListChoices(SH-F3)  > :
```

5. Press Tab to accept the default filename, or enter a different filename and press Tab.

6. Use one of the following methods to specify the return type of the value that this procedure returns:

   - Press Tab to accept the default of "integer."
   - Enter a standard Ingres data type or a record type that you have created for this application and press Tab.
   - Enter none to indicate no return value and Press Tab.

   The cursor skips the Nullable field; you cannot change the default value of "no."

   The cursor moves to the Symbol field. This value specifies the name that the host operating system linker uses to call the procedure. Vision assumes that the symbol name is the same as the procedure name.

7. Press Tab to accept this default symbol name, or enter a new name.

8. Select End to return to the Application Flow Diagram.

Specify the Link Options File

If your 3GL procedure is in a library, you must specify the link options file through which Vision is to call the procedure. You do this on the Application Defaults Window.

To specify the link options file

1. On the Vision Applications Catalog window, position the cursor on the name of the application that contains the 3GL library procedure.
2. Select MoreInfo from the menu.

   Vision displays the MoreInfo About an Application window.
3. Select Defaults from the menu.

   Vision displays the Application Defaults window.
4. Move the cursor to the Link-options Filename field.
5. Specify the name of the link options file and select OK.
6. Select Save on the MoreInfo About an Application window.
7. Select End to return to the Applications Catalog window.

Define a Database Procedure

You can include in a Vision application a procedure composed of SQL statements stored in the database. You use the SQL create procedure statement to create the procedure before you define it in Vision.

See the *SQL Reference Guide* for more information about creating database procedures. You cannot use an SQL database procedure if you are accessing a non-Ingres database through an Enterprise Access product (formerly Gateway).

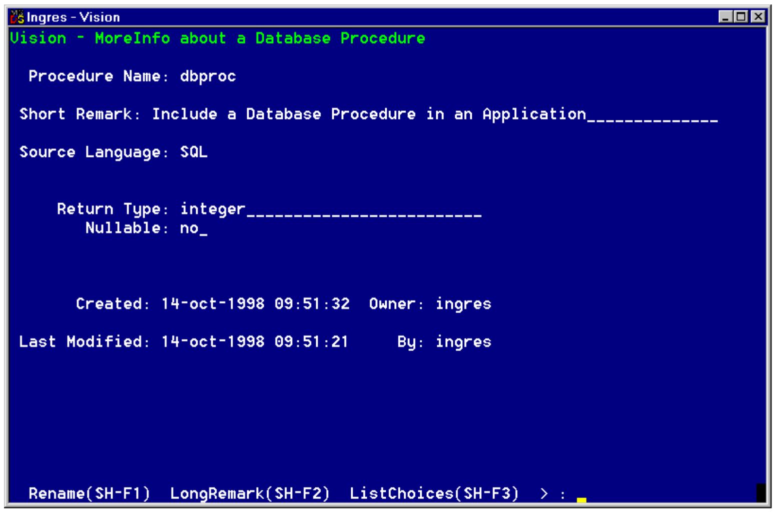
To define a database procedure

1. Follow Steps 1 through 8 in Create a New Frame (see page 98). Specify "DB Proc" as the type.

   Vision redisplays the Application Flow Diagram with the new database procedure inserted.
2. Select MoreInfo to display the MoreInfo About a Database Procedure Window, as shown in the following figure.

   The window displays the procedure name and description you just entered, as well as when and by whom the frame was created. When the window is displayed, the cursor is on the Return Type field.

3. Specify in one of the following ways the return type of the value that this procedure returns:
   - Press Tab to accept the default of "integer."
   - Enter a standard Ingres data type or a record type that you have created for this application and press Tab.
   - Enter none to indicate no return value and press Tab.



```
Vision - MoreInfo about a Database Procedure

  Procedure Name: dbproc

 Short Remark: Include a Database Procedure in an Application______________

 Source Language: SQL


     Return Type: integer_________________________
        Nullable: no_


       Created: 14-oct-1998 09:51:32  Owner: ingres

 Last Modified: 14-oct-1998 09:51:21     By: ingres





  Rename(SH-F1)  LongRemark(SH-F2)  ListChoices(SH-F3)  > :
```

   The cursor moves to the Nullable field. The default value of "no" indicates that this procedure cannot return a null value.

4. Press Tab to accept the default, or type yes and press Tab to permit a nullable return value.
5. Change the descriptive Short Remark that you entered when you created the procedure in Vision.
6. When you are done, select End to return to the Application Flow Diagram.

Create and Edit the Source Code for a 4GL or 3GL Procedure

After you define a 3GL or 4GL procedure, you can call the system editor from Vision to create or edit the source code for the procedure. You can create the source code file at any time before you run the procedure.

You cannot access a database procedure directly from Vision.

To create or edit the source code for a 4GL or 3GL procedure

1. Select the procedure as the current frame in the Application Flow Diagram.
2. Select Edit from the menu.

3. Select Source Code File to open the source code file for the procedure.

   Vision calls the system editor and opens the file you specified when you defined the procedure. You can write new source code or edit existing code.

4. When you are done, save the file.

   When you exit the system editor, Vision returns you to the Application Flow Diagram.

Change the Definition of a Frame or Procedure

After you have defined a frame or procedure, you can change some of the specifications you entered; for example, to use a different report for a Report frame.

To change the definition of a frame or procedure

1. Select the frame or procedure as the current frame in the Application Flow Diagram.
2. Select MoreInfo from the menu.

   Vision displays the MoreInfo About a Frame or MoreInfo About a Procedure window. The window contains the current information about the frame or procedure.

3. Change any of the specifications as desired. Refer to the appropriate section above to see the changes that are allowed for each frame or procedure type.

   Select End to return to the Application Flow Diagram.

Chapter 8: Using Vision Advanced Features

This section contains the following topics:

This chapter describes the advanced functions that Vision provides. When you develop Vision applications, you can use these functions to:

- Specify more complex queries
- Make your applications easier to use
- Make your applications more efficient

Vision advanced features include:

- *Parameters* to pass values between frames
- *Local variables* to define hidden fields for a frame
- *Local procedures* to define 4GL procedures for a frame
- *Global components—constants* and *variables* to set values throughout an entire application, or *record types* to define your own data types
- *Escape code* that you write to perform additional functions at specific points when you run a frame

  To write escape code, you need to be familiar with 4GL. The 4GL part of this guide provides a comprehensive guide to the 4GL statements and their syntax.

Note: This chapter contains a separate section describing how to use each of the advanced features. However, you can use several of these features together; for example, by writing escape code to tell Vision how to use a local variable. To combine advanced features, refer to the specific sections of this chapter that describe those functions.

Passing Parameters Between Frames

You can use the current values from a parent frame to set the values for fields on a child frame. A value that you pass between frames in this way is called a *parameter*. Parameters let you use the same data with different frames.

When you use parameters, Vision lets you move data from the parent frame to the child frame so that the user:

- Does not have to re-enter the information
- Is ensured of working with the correct data

  For example, a user might display a record on a Browse frame about a specific customer order, then call an Update frame to change that record. Disable user qualification processing and pass values from the Browse frame to the Update frame. When the user calls the Update frame:

  – The Update frame displays the customer record that currently appears on the Browse frame.

  – Vision automatically increases the price of each item in the order by 10% on the Update frame (in effect, passing the value of the price field plus 10%).

The figure in Pass a Parameter to a Frame (see page 180) shows how to specify these parameters and provides additional examples.

Guidelines for Passing Parameters

Follow these rules and guidelines when passing values between frames:

- You can pass parameters *from* Menu, Append, Browse, or Update frames only (because these are the only frames that can have children in a Vision application).
- The fields between which you pass the parameters can be either hidden or displayed.
- The parameter must be in the form of one of these values from the parent frame:

  – The name of a field on the form

  – A hidden field

    This can be either a local or global variable. Creating local and global variables is discussed later in this chapter.

  – An expression

    The discussion of expressions in Using Expressions in Frame Definitions (see page 155) also applies to parameters. For more comprehensive information on expressions, see the 4GL part of this guide.

  - A database query

    For example, passing the following parameter from the parent frame:

    ```
    select name from customers
    where customer_no = :customer_no
    ```

    displays on the child frame the name of the customer whose customer number appears on the parent frame.

    See the 4GL reference part of this guide for more information about passing queries between frames.

  - A local procedure

    Use a local procedure as a parameter when the return value of the procedure is the correct data type to pass as a parameter. For example, if you have a procedure "myproc" that returns an integer value, specifying the parameter:

    ```
    myproc()
    ```

    passes the integer value to the child frame.

    Local procedures are discussed in more detail in Using Local Procedures (see page 187).

- Be sure that the values on the parent and child frames have consistent data types.

  For example, you cannot pass a value from an integer field into a character field. If Vision cannot pass a parameter, you get a runtime error when you call the child frame.

- Vision passes parameters *by value* as the default method.

  A parameter passed by value is not affected by any changes made to the value on the child frame. For example, if you pass a parameter with a value of 2 and then change that value to 4 on the called frame, the value still is 2 on the calling frame when you return to it.

  You also can pass a parameter *by reference*. A parameter passed by reference is changed in the calling frame if its value is changed in the called frame. In the example above, the value on the calling frame is 4 when you return to it.

  To pass a parameter by reference, include the keyword byref before the specification of the value from the parent frame. For example:

  ```
  byref (quantity)
  ```

  You cannot pass a global variable as a parameter by reference.

- The value that you want to pass must be current for the parent frame. If the value is removed—for example, by clearing the form or saving it to the database—it is not passed to the child frame.

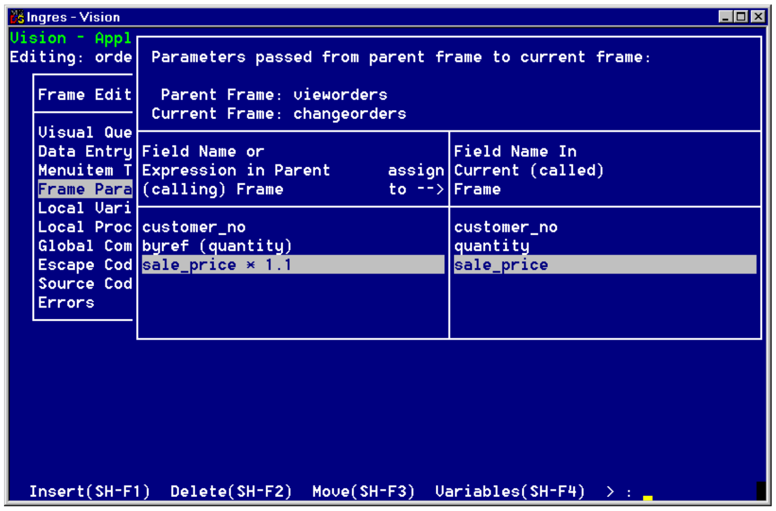
- To pass parameters from a parent frame to any number of its child frames, you must specify the parameter separately for each child to which it is to be passed.
- You cannot pass a value from a simple field to a column in a table field. If Vision cannot pass a parameter, you get a runtime error when you call the child frame.

Pass a Parameter to a Frame

To pass a parameter to a child frame

1. In the Application Flow Diagram, select as the current frame the frame to which you are passing the parameters.
2. Select Edit from the menu.
3. Select Frame Parameters from the "Frame Edit Options" list.

   Vision displays the pop-up window for passing parameters as shown in the following figure:



```
Vision - Appl
Editing: orde  Parameters passed from parent frame to current frame:

 Frame Edit     Parent Frame: vieworders
               Current Frame: changeorders
 Visual Que
 Data Entry Field Name or                      Field Name In
 Menuitem T Expression in Parent        assign Current (called)
 Frame Para (calling) Frame             to --> Frame
 Local Vari
 Local Proc customer_no                        customer_no
 Global Com byref (quantity)                   quantity
 Escape Cod sale_price * 1.1                   sale_price
 Source Cod
 Errors

  Insert(SH-F1)  Delete(SH-F2)  Move(SH-F3)  Variables(SH-F4)  > :
```

4. Use one of these methods to specify the value from the parent frame:
   - Enter a fieldname or expression.

     Do not include a colon (:) with the fieldname.
   - Use the ListChoices operation as follows:
     a. Select ListChoices from the menu.
     b. Select an item from the list of available types of values: Form Fields, Local Variables, Global Variables and Global Constants.

        Vision displays a list of values available from the parent frame.
     c. Select a value from the list.

        Vision enters the value into the parent frame side of the window.

     To use this value as part of an expression, enter the rest of the expression—for example, an arithmetic operation such as "*1.1"—in the window.

     To pass this parameter by reference, enter:

     ```
     byref (parameter)
     ```

   You also can use the Variables operation to create a new local or global component to specify in a parameter.
5. Press Tab.
6. Enter the name of the field on the child frame that is to receive the value.

   Do not include a colon (:) with the fieldname.

   You can specify this value as in Step 4. If you use the ListChoices operation, Vision displays the relevant values for the child frame.
7. Repeat Steps 4 through 6 for each parameter you want to specify.

   You can use the up and down arrow keys to move to a new line, or select the Insert operation to place a parameter on a specific line. You can delete a parameter or change its location in the pop-up window by using the Delete and Move operations, respectively.
8. When you have indicated all the values you want to pass, select End to save your parameters.

To pass parameters to other children of the parent frame, select each child frame in turn and repeat the above procedure.

Using Local Variables

In addition to the fields that appear on a frame's form, you can create hidden fields for Append, Browse, Update, and Menu frames. A hidden field holds information that the user cannot see.

In Vision, hidden fields are known as *local variables*, because you create them and set their values locally for an individual frame. You can create a maximum of fifty local components (local variables and local procedures) for each frame.

After you create a local variable as described below, you write escape code to define its value. You then can use the local variable:

- In a query restriction on a Browse or Update frame
- In an assigned value for a field on an Append frame
- You can assign a value to a non-displayed field on an Append frame without having to define it as a local variable.
- In a parameter that you pass to another frame
- In additional escape code

Parameters and escape codes are discussed in other sections of this chapter.

For example, create a local variable called "h_date" to keep track of the current date and time. The following escape code uses standard Ingres functions to define its value as the current time:

```
h_date = date ('now');
```

Use this local variable in escape code to record each time that a user changes a record on an Update frame. See How Writing Escape Code Works (see page 210) for more information about using escape code in your applications.

You also can create local variables based on existing columns or table-field columns in the Master or Detail table for a frame. For more information, see Change the Default Form Display (see page 122).

Naming Guidelines for Local Variables

The names that you give to local variables must follow the standard naming conventions. The following naming guidelines also apply to local variables:

- A general convention is to begin the name of a local variable with "h_" to designate it as hidden field; for example, *h_date* in the example above.
- The name of a local variable can be up to 65 characters in length.

- For a hidden column in a table field, include the table-field name at the beginning of the local variable name; for example, *detailtbl.h_custno*.
- Do not begin any variable names with the string "ii", because this prefix is reserved for use by Ingres. Vision adds the string "iih_" to the beginning of all variable names when it generates the code for a frame.
- A local variable cannot have the same name as a displayed column in the Master or Detail table used for this frame.
- You can define a local variable with the same name as a non-displayed column in the Master or Detail table (except for a non-displayed column with an assigned value on an Append frame).
- A local variable cannot have the same name as a local procedure.

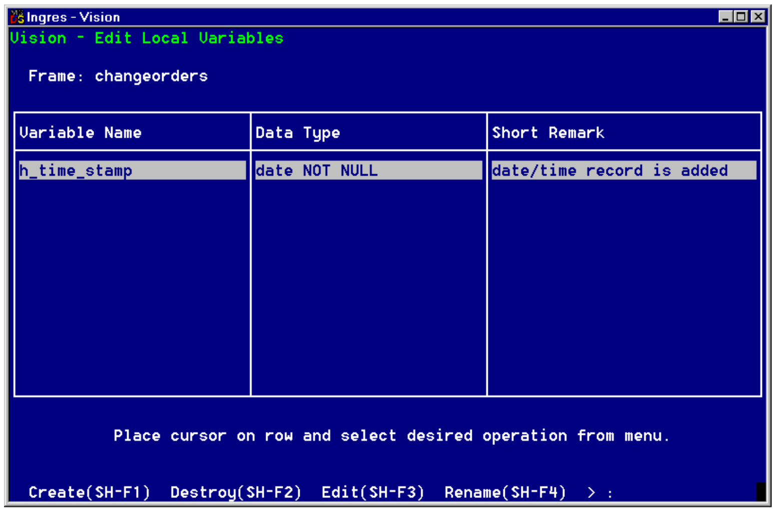
Create a Local Variable

To create a local variable for an Append, Browse, or Update frame

1. Select the frame as the current frame in the Application Flow Diagram.
2. Select Edit from menu.

   You also can select Edit from the Visual Query Editor. Select Local Variables from the menu.

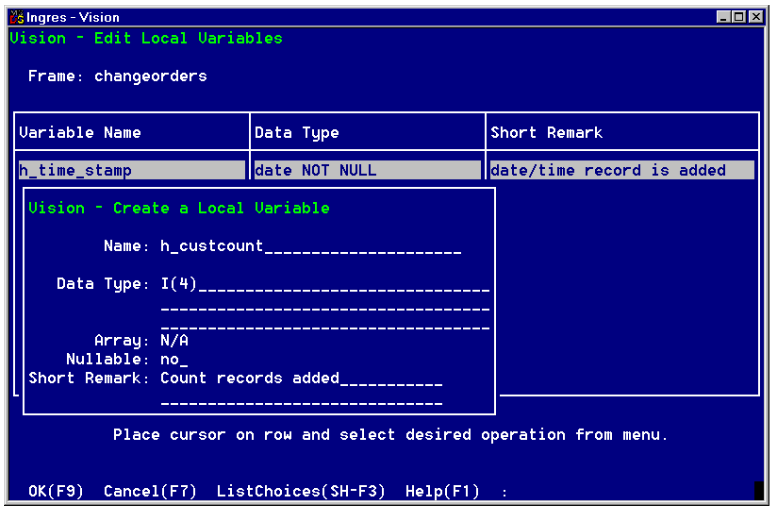
   Vision displays the Edit Local Variables Window as shown in the following figure:



```
Ingres - Vision
Vision - Edit Local Variables

  Frame: changeorders

 Variable Name            Data Type                Short Remark
 h_time_stamp             date NOT NULL            date/time record is added

            Place cursor on row and select desired operation from menu.

  Create(SH-F1)  Destroy(SH-F2)  Edit(SH-F3)  Rename(SH-F4)  > :
```

3. Select Create from the menu.

   Vision displays the Create a Local Variable pop-up window as shown in the following figure:



```
Ingres - Vision
Vision - Edit Local Variables

  Frame: changeorders

 Variable Name            Data Type                 Short Remark
 h_time_stamp             date NOT NULL             date/time record is added
   Vision - Create a Local Variable

           Name: h_custcount_____________________

      Data Type: I(4)________________________________
                 ____________________________________
                 ____________________________________
          Array: N/A
       Nullable: no_
   Short Remark: Count records added___________
                 ______________________________

           Place cursor on row and select desired operation from menu.

  OK(F9)  Cancel(F7)  ListChoices(SH-F3)  Help(F1)  :
```

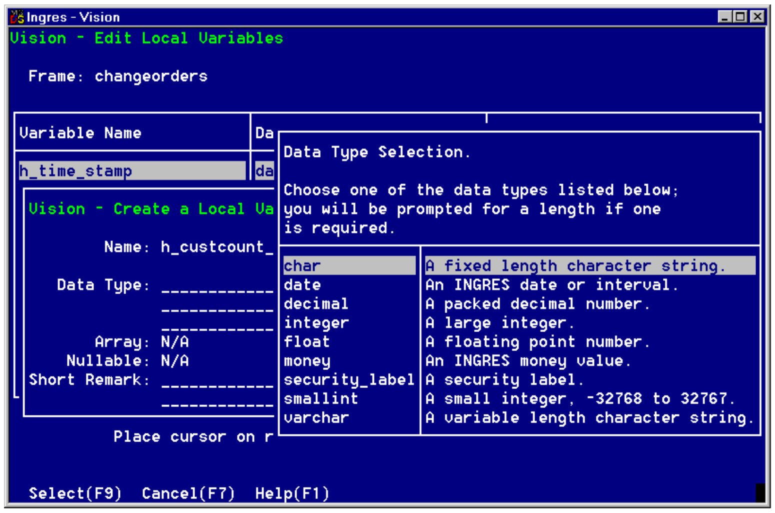
4. Enter a name for the local variable according to the guidelines given above.

5. Press Tab.

6. Specify the data type for the local variable in either of these ways:

   - Enter a valid Ingres data type or the name of a record type that you created previously for this application.
   - Use the ListChoices operation.

   Vision displays a list of valid data types as shown in the following figure:



```
Ingres - Vision
Vision - Edit Local Variables

  Frame: changeorders

 Variable Name            Da
                            Data Type Selection.
 h_time_stamp             da
                            Choose one of the data types listed below;
   Vision - Create a Local Va you will be prompted for a length if one
                            is required.
           Name: h_custcount_
                            char          A fixed length character string.
      Data Type: ____________ date          An INGRES date or interval.
                 ____________ decimal       A packed decimal number.
                 ____________ integer       A large integer.
          Array: N/A          float         A floating point number.
       Nullable: N/A          money         An INGRES money value.
   Short Remark: ____________ security_label A security label.
                 ____________ smallint      A small integer, -32768 to 32767.
                            varchar       A variable length character string.
           Place cursor on r

  Select(F9)  Cancel(F7)  Help(F1)
```

Be sure the data type is consistent with the function of the local variable. For example, if the value of the variable is a name, use a character data type.

7. Press Tab.

   If you have specified a record type in the Type field, the Array field is displayed. Enter yes to indicate the variable is an array or no to indicate the variable is a single record.

   If you have specified an Ingres data type in the Type field, the Nullable field is displayed. The default value of yes indicates that the variable can accept a null value. Enter no to make the variable not nullable.

8. Press Tab.

9. Enter a Short Remark to describe this variable.

   This remark appears in the Edit Local Variables window. It also appears as a comment in the 4GL code that Vision generates for this frame.Select OK from the menu.

   Vision saves your local variable and places it in the Edit Local Variables window.

10. Repeat Steps 4 through 12 to create additional local variables, or select End to return to the window in which you were working.

Assign a Value to a Local Variable

Vision declares any local variables in the initialize block when it generates the 4GL code for a frame. Vision sets the initial value of local variables to a blank string for character data types and zero for numeric data types.

After you define a local variable, you can change this default initial value.

To assign a value to a local variable

1. Write a 4GL assignment statement.

2. Place this statement as a Form-Entry escape code for the frame.

See How Writing Escape Code Works (see page 210) for more information.

Edit a Local Variable Definition

After you create a local variable, you can edit its definition to change its data type or nullability.

To modify a local variable definition

1. Display the Edit Local Variables window as described in Create a Local Variable (see page 183).
2. Position the cursor on the name of the variable you want to edit.
3. Select Edit.

   Vision displays the Edit a Local Variable pop-up window. This window contains the information you entered when you created the variable.
4. Specify a new data type or change the nullability as described in Create a Local Variable (see page 183).
5. Select OK to save your changes.

   Vision displays the new definition in the Edit Local Variables window.
6. Select End to return to the window in which you were working.

Rename a Local Variable

You can change the name of a local variable you have defined. However, you also must change the name at each place you use the variable throughout the application, including any escape code.

To rename a local variable

1. Display the Edit Local Variables window as described in Create a Local Variable (see page 183).
2. Position the cursor on the name of the variable.
3. Select Rename from the menu.

   Vision prompts you for the new name.
4. Enter a new name for the variable.
5. Press Return.

   The new name of the variable appears in the Edit Local Variables window.

Destroy a Local Variable

You can destroy a local variable that you no longer are using for a frame. However, be sure that you delete all references to the variable in the application, including any escape code, or Vision cannot compile and run your application.

To destroy a local variable

1. Display the Edit Local Variables window as described in Create a Local Variable (see page 183).
2. Position the cursor on the name of the variable you want to destroy.
3. Select Destroy from the menu.

   Vision asks you to confirm that you want to destroy the variable you have selected.
4. Enter y and press Return.

   Vision destroys the variable and removes it from the Edit Local Variables window display.

Any local variables for a frame are destroyed automatically if you destroy the frame.

Using Local Procedures

You can create local procedures for Append, Browse, Update, and Menu frames. A local procedure is a 4GL procedure that you can create and use for an individual frame. You can create a maximum of fifty local components (local procedures and local variables).

After you create a local procedure as described below, you write 4GL source code to define its value. You then can use the local procedure:

- In an assigned value for a field on an Append frame
- In a parameter that you pass to another frame
- In escape code

Guidelines for Naming Local Procedures

The names that you give to local procedures must follow the standard naming conventions. The following naming guidelines also apply to local procedures:

- The name of a local procedure can be up to 65 characters in length.
- Do not begin any procedure names with the string "ii".
- A local procedure cannot have the same name as a local variable.

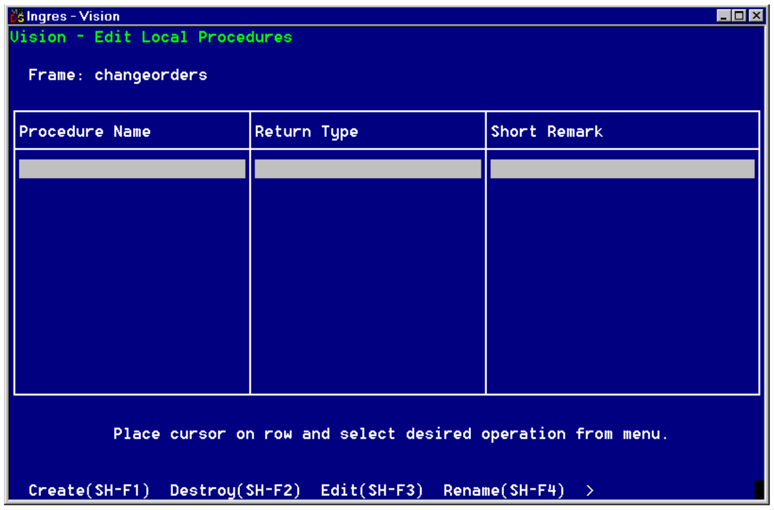
Create a Local Procedure

To create a local procedure for an Append, Browse, Update, or Menu frame

1. Select the frame as the current frame in the Application Flow Diagram.
2. Select Edit from menu.

   You also can select Edit from the Visual Query Editor.
3. Select Local Procedures from the menu.

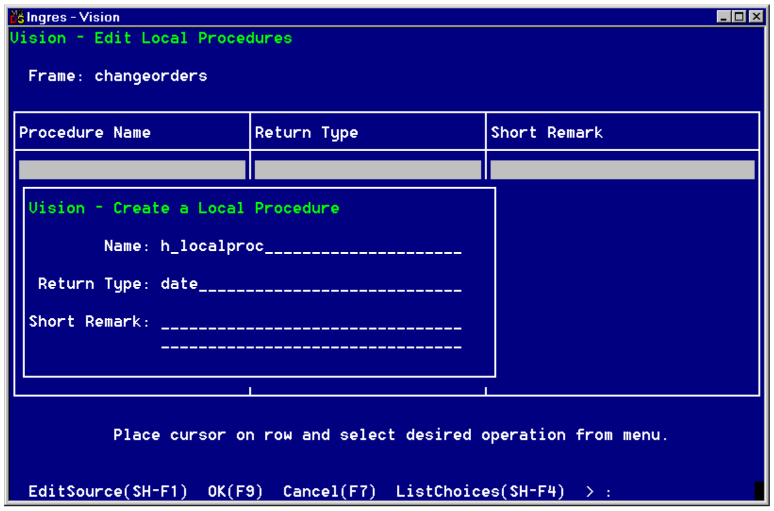
   Vision displays the Edit Local Procedures window as shown in the following figure:



```
Vision - Edit Local Procedures

  Frame: changeorders

 Procedure Name            Return Type               Short Remark

           Place cursor on row and select desired operation from menu.

  Create(SH-F1)  Destroy(SH-F2)  Edit(SH-F3)  Rename(SH-F4)  >
```

4. Select Create from the menu.

   Vision displays the Create a Local Procedure pop-up window as shown in the following figure:



```
Ingres - Vision
Vision - Edit Local Procedures

  Frame: changeorders

 Procedure Name          | Return Type            | Short Remark

  Vision - Create a Local Procedure

          Name: h_localproc____________________

   Return Type: date___________________________

  Short Remark: _______________________________
                _______________________________

           Place cursor on row and select desired operation from menu.

  EditSource(SH-F1)  OK(F9)  Cancel(F7)  ListChoices(SH-F4)  > :
```

5. Enter a name for the local procedure according to the guidelines given above.

6. Press Tab.

7. Specify the data type for the return value of the local procedure in either of these ways:

   - Enter a valid Ingres data type or "none" if your procedure does not return a value. Local procedures can only return simple data types, not record types or complex data types.
   - Use the ListChoices operation.

     Vision displays a list of valid data types. Select a data type from the list. If you select a data type that requires a length, Vision displays a popup where you can enter the length.

   Be sure the data type is consistent with the value returned from the local procedure. For example, if the return value of the procedure is a name, use a character data type.

8. Press Tab.

9. Enter a Short Remark to describe this procedure.

   This remark appears in the Edit Local Procedures window. It also appears as a comment in the 4GL code that Vision generates for this frame.

10. To create only the procedure definition, select OK. Vision saves your local procedure and places it in the Edit Local Procedures window.

    To enter the code for the procedure, select EditSource from the menu. Proceed with Step 5 in Enter Local Procedure Code (see page 190).

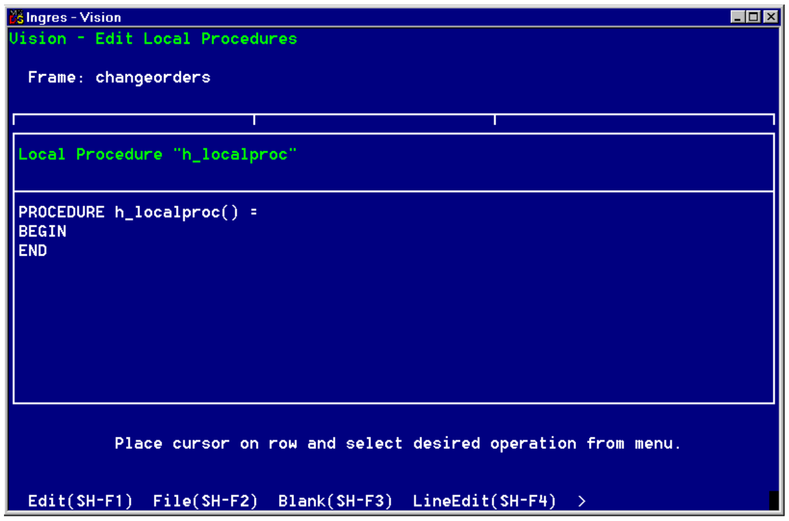
11. Repeat Steps 4 through 10 to create additional local procedures, or select End to return to the window in which you were working.

Enter Local Procedure Code

To enter or change the code for a local procedure

1. Display the Edit Local Procedures window as described in the Creating a Local Procedure section.
2. Position the cursor on the name of the procedure you want to edit.
3. Select Edit from menu.
4. Select EditSource from the menu.

   Vision displays the local procedure source code popup. Before you enter any code for a procedure, the popup shows the code for an empty procedure. See the following figure:



```
Ingres - Vision
Vision - Edit Local Procedures

  Frame: changeorders

 Local Procedure "h_localproc"

 PROCEDURE h_localproc() =
 BEGIN
 END

           Place cursor on row and select desired operation from menu.

  Edit(SH-F1)  File(SH-F2)  Blank(SH-F3)  LineEdit(SH-F4)  >
```

5. Edit the source code in one of the following ways:
   - Directly in the local procedures pop-up window
   - With the default system editor
   - From an external file

   Each of these methods is described in its own section below.
6. From the local procedures popup, select Save.

7. Select End to return to the Edit Local Procedures window.
8. Repeat Steps 2 through 7 to edit additional local procedures, or select End to return to the window in which you were working.

How You Can Enter Local Procedure Code Directly in the Window

When Vision displays the pop-up window for writing local procedure code, you can enter your code directly in the window. The standard Ingres window editing keys and functions are available.

The following additional edit operations are available on the local procedure pop-up window:

Edit

Calls the system editor and opens a file for you to write your local procedure code, as described in Enter Code Using the System Editor

Blank

Clears the window. Be aware that there is no "undelete" function to restore text after you have cleared the window.

LineEdit

Displays a submenu with these line editing functions:

InsertLine

Inserts a blank line above the line on which the cursor is positioned

DeleteLine

Deletes the line on which the cursor is positioned

SplitLine

Divides a line into two lines at the point where the cursor is positioned

JoinLines

Moves the next line to the end of the line on which the cursor is positioned

Variables

Lets you create or edit local variables or global components to use in your local procedure code

Save

Saves the local procedure code you have entered

Enter Code Using the System Editor

You can use your system editor to write your local procedure code, rather than using the line edit functions. Vision uses the default system editor.

See your Ingres system administrator if you want to change this default. If you are using a PC, see the system administrator of the remote node.

To write local procedure code using your system editor

1. When Vision displays the pop-up window for entering local procedure code, select Edit from the menu.

   Vision calls the system editor and opens a file.

2. Enter your code in the window, using the standard editor operations.
3. When you are finished entering your code, save your file and exit from the editor.

   Vision places the text from the file into the local procedure code pop-up window.

   To edit the file containing this code, select Edit again. Vision redisplays the file.

Enter Code Using External Files

When editing local procedure code, you can use external text files to:

- Write text from the window to a file.
- Read text from a file to the window.

Using an external text file is especially convenient if you are using the same or similar local procedure code in more than one place. You can write the code to a file when you first create it, then read in this file for additional instances of local procedure code.

When you write local procedure code to an external file, you simply are using it to store your code. Vision does not access the file itself when it runs a frame containing the local procedure code.

You can store a basic version of the code in the file, then call the file and change the code as needed each time you use it.

To write local procedure code to a file

1. Enter the code in the local procedure code pop-up window.
2. Select File from the menu.
3. Select WriteFile from the submenu.

   Vision prompts you for a file name.

4. Enter the name of the file in which you want to save your local procedure code.

   If the file is not in the current directory, be sure to enter the entire directory path.

5. Press Return.

   Vision saves your text in the file and returns you to the local procedure code pop-up window.

To read local procedure code from a file

1. Display the pop-up window for entering local procedure code.
2. Select File from the menu.
3. Select ReadFile from the submenu.

   Vision prompts you for the name of the file.

   If the file is not in the current directory, be sure to enter the entire directory path.

4. Enter the name of the file containing the local procedure code and press Return.

   Vision writes the file to the window. If any text is in the window, Vision places the contents of the file above the line on which the cursor is positioned.

If you make any changes to the file text while it is displayed in the window, you can either write the new text back to the file or create a new file.

Edit a Local Procedure Definition

After you create a local procedure, you can edit its definition to change the return data type, or the short remark. You can also change the procedure itself by editing the 4GL source code.

To modify a local procedure definition

1. Display the Edit Local Procedures window as described in Create a Local Procedure (see page 188).
2. Position the cursor on the name of the procedure you want to edit.
3. Select Edit.

   Vision displays the Edit a Local Procedure pop-up window. This window contains the information you entered when you created the variable.

4. Specify a new return data type, or a new short remark, as described in Create a Local Procedure (see page 188).
5. Select OK to save your changes. If you do not want to save your changes, select Cancel.

   Vision displays the new definition in the Edit Local Procedures window.

Rename a Local Procedure

You can change the name of a local procedure you have defined. You must change the name in the source code of the procedure as well as in the Edit Local Procedures window. You also must change the name at each place you use the procedure throughout the application, including any escape code.

To rename a local procedure

1. Display the Edit Local Procedures window as described in Create a Local Procedure. (see page 188)
2. Position the cursor on the name of the procedure.
3. Select Rename from the menu.

   Vision prompts you for the new name.
4. Enter a new name for the procedure.
5. Press Return.

   Vision displays a warning that you must change the procedure name in the text of the procedure.
6. Press Return.

   The new name of the procedure appears in the Edit Local Procedures window.
7. With the cursor on the new name, select Edit from the menu.
8. Select EditSource.
9. Change the procedure name in the first line of code:

   ```
   PROCEDURE newname() =
   ```
10. Select Save.
11. Select End.

    The new name appears in the Edit Local Procedures window.

Destroy a Local Procedure

You can destroy a local procedure that you no longer are using for a frame. However, be sure that you delete all references to the procedure in the application, including any escape code, or Vision cannot compile and run your application.

To destroy a local procedure

1. Display the Edit Local Procedures window as described in Create a Local Procedure (see page 188).
2. Position the cursor on the name of the procedure you want to destroy.
3. Select Destroy from the menu.

   Vision asks you to confirm that you want to destroy the procedure you have selected.
4. Enter y and press Return.

   Vision destroys the procedure and removes it from the Edit Local Procedures window display.

Any local procedures for a frame are destroyed automatically if you destroy the frame.

Global Components

You can specify values to use throughout an entire application. These values are known as *global* components. There are three types of global components you can create in Vision:

- *Constants* to use the same value throughout an application
- *Variables* to set values for a whole application
- *Record types* to define your own data types

Because these values apply to an entire application, you generally can create and use them from any frame. The following sections describe how to create each of these global components and how to use them in your application.

Using Global Constants

You can define a specific value as a *global constant* to use at various points throughout an application. You then can use that constant by itself or as part of a 4GL expression in any of these ways:

- To assign values to fields in Append frames
- To specify query restrictions for Browse and Update frames
- In a parameter that you pass between frames
- In 4GL escape code

Global constants are convenient for values that can change over the lifetime of an application. For example, you might need to include a value for a sales tax on several frames. Define a constant called "sales_tax" and set its value to the current tax rate.

If the tax rate changes, you need to reset the value for only the single constant you have defined. The next time you run the application, Vision applies the new value for "sales_tax" wherever it appears.

You can define constants as character strings or numerical values. For example, you can have a message that appears on various frames. By defining the text of the message as a constant, you can easily change the message throughout the application.

See Use Alternate Sets of Global Constants (see page 256).

Define a Global Constant

To define a global constant

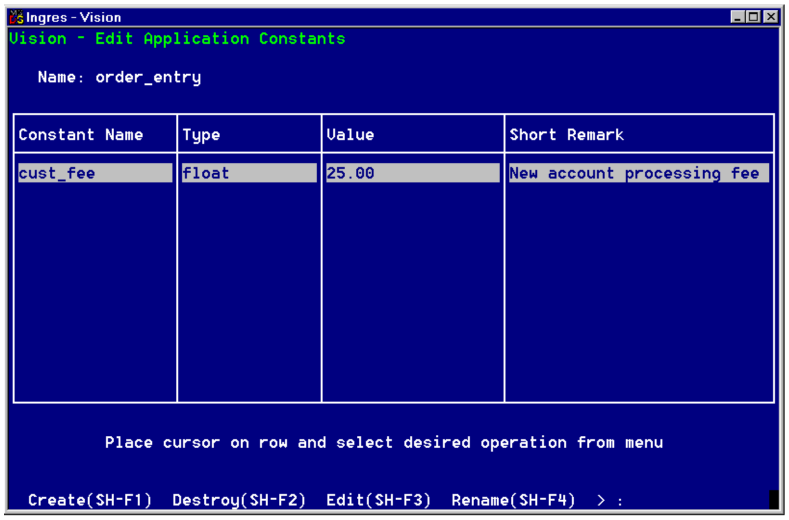
1. Select Edit from the menu in the Application Flow Diagram Editor or Visual Query Editor window.

   Because the constant definition applies to the entire application, you can select any frame as the current frame.

2. Select Global Components from the list of options.

3. Select Constants from the list of global components.

   Vision displays the Edit Application Constants Window as shown in the following figure:



```
Ingres - Vision
Vision - Edit Application Constants

   Name: order_entry

 Constant Name    Type            Value               Short Remark
 cust_fee         float           25.00               New account processing fee

          Place cursor on row and select desired operation from menu

  Create(SH-F1)  Destroy(SH-F2)  Edit(SH-F3)  Rename(SH-F4)  > :
```

4. Select Create from the menu.

   Vision displays the Create an Application Constant pop-up window as shown in the following figure.

5. Enter a name for the constant.

6. Press Tab.

7. Enter an optional Short Remark to describe the constant.

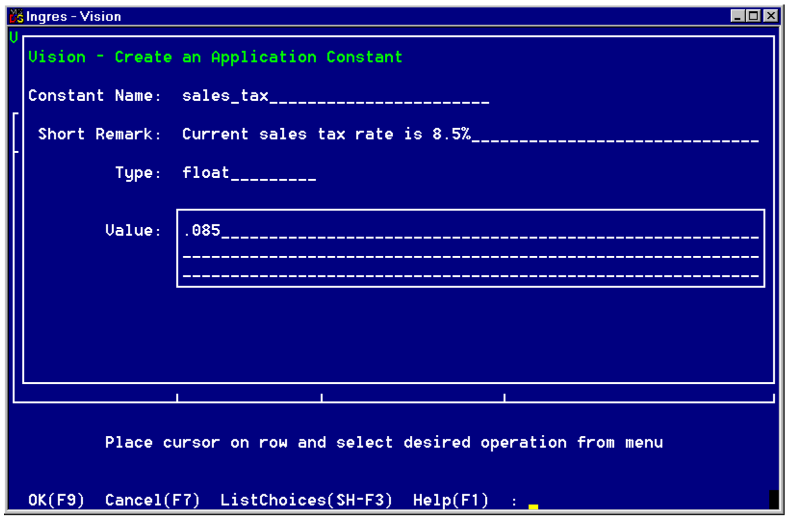
   This description appears in the Edit Application Constants window. It also appears as a comment in the code that Vision generates to define the constant.

8. Press Tab.

9. Specify a data type in either of these ways:

   - Enter a valid Ingres data type. If the type requires a length, specify the length in parentheses. If you do not specify the length, Ingres calculates the length based on the value you enter in the Value field.
   - Use the ListChoices operation.

     Vision displays a list of valid Ingres data types. If the type you select requires a length, enter the length in parentheses after Vision places the data type in the Type field. If you do not specify the length, Ingres calculates the length based on the value you enter in the Value field.



```
Vision - Create an Application Constant

Constant Name:  sales_tax_______________________

 Short Remark:  Current sales tax rate is 8.5%______________________________

         Type:  float_________

        Value:  .085____________________________________________________________
                ________________________________________________________________
                ________________________________________________________________

        Place cursor on row and select desired operation from menu

OK(F9)  Cancel(F7)  ListChoices(SH-F3)  Help(F1)  :
```

     For example, to indicate a char(20) data type, select "char" then add "(20)".

   Be sure the data type is consistent with the function of the constant. For example, if the value of the constant is a percentage, use a float data type.

10. Press Tab.
11. Enter a value for the constant.

    Be sure that the value is consistent with the data type.

12. Select OK from the menu.

    If the value for the constant is longer than the length you specified in the type field, Vision prompts you to truncate the value or to increase the length of the data type. Select the appropriate response.

    Vision displays the new constant in the Edit Application Constants window.

13. Select End to return to the window in which you were working.

Edit a Global Constant Definition

You can change the value of a global constant or edit the Short Remark that describes it. You can also modify the length of a constant's data type, but you cannot change a constant's data type. You must define a new constant to change the data type.

To change the value, description, or data type length of a global constant

1. Display the Edit Application Constants window as described in Steps 1 through 3 for Define a Global Constant (see page 196).
2. Position the cursor on the name of the constant you want to change.
3. Select Edit from the menu.

   Vision displays the Edit a Constant Definition window with the cursor positioned on the Short Remark field. The field contains an optional brief description of the constant.
4. Edit the Short Remark.
5. Press Tab.
6. Change the data type length, if necessary. If you specify a data type without a length, Ingres calculates the length based on the value you entered for the constant. You cannot change the data type for an existing constant, only the length.
7. Press Tab.
8. Enter a new value for the constant.

   Be sure that the value is consistent with the data type.
9. Select End.

   If the value for the constant is longer than the length you specified in the type field, Vision prompts you to truncate the value or to increase the length of the data type. Select the appropriate response.

   Vision saves your changes and displays the new value, data type length, or description on the Edit Application Constants window.
10. Select End again to return to the window in which you were working.

Rename a Global Constant

To change the name of a global constant you have defined, be sure to change the name at each place you use the constant in the application.

To rename a global constant

1. Display the Edit Application Constants window as described in Steps 1 through 3 for Define a Global Constant (see page 196).
2. Position the cursor on the name of the constant you want to rename.
3. Select Rename from the menu.

   Vision prompts you for the new name.
4. Enter a new name for the constant.
5. Press Return.

   The new name of the constant appears in the Edit Application Constants window.

Destroy a Global Constant

To destroy a global constant that you no longer are using, be sure that you delete all uses of the constant in the application, or Vision cannot compile and run your application.

To destroy a global constant

1. Display the Edit Application Constants window as described in Steps 1 through 3 Define a Global Constant (see page 196).
2. Position the cursor on the name of the constant.
3. Select Destroy from the menu.

   Vision asks you to confirm that you want to destroy the constant you have selected.
4. Enter y and press Return Vision removes the constant from the application constants catalog.

Using Global Variables

In addition to constants, you can create variables—called *global variables*—to use throughout an application. You can use the variable in any of these ways:

- In an expression to assign values on an Append frame
- In an expression to specify a query restriction on a Browse or Update frame
- As part of a parameter passed between frames
- As a local variable on a frame

  Note: Global variables can be used wherever local variables are used. However, do not replace all local variables with global variables; this increases compilation time because global variables are stored in the system catalogs. You must use local variables instead of global variables wherever possible. Do not use a global variable unless the variable is used in several frames, and you do not want to pass it as a parameter.

- In 4GL escape code

  For example, create a global variable called "user_id" to store users' passwords. Write escape code to prompt users for their passwords when they start the application and check the value of "user_id" against a table of authorized users before allowing access to the application.

  See How Writing Escape Code Works (see page 210) for an example of this code.

Defining a Global Variable

To define a global variable

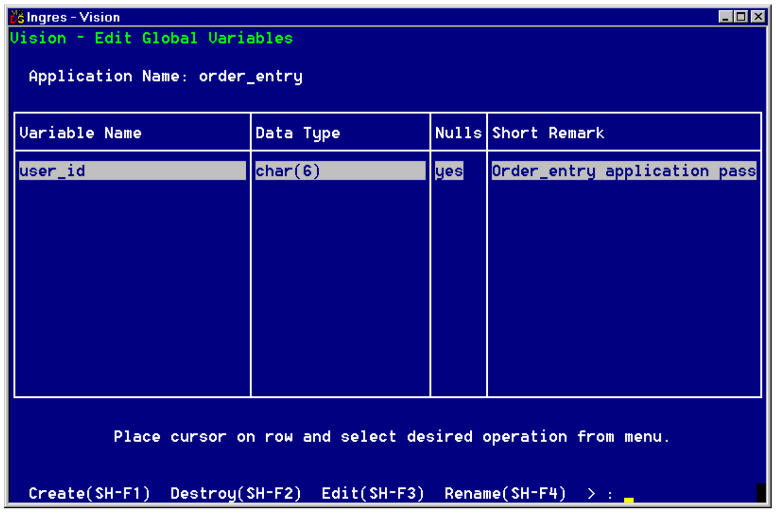
1. Select Edit from the menu in the Application Flow Diagram Editor or Visual Query Editor window.

   Because the variable definition applies to the entire application, you can select any frame as the current frame.

2. Select Global Components from the list of options.

3. Select Variables from the list of global components.

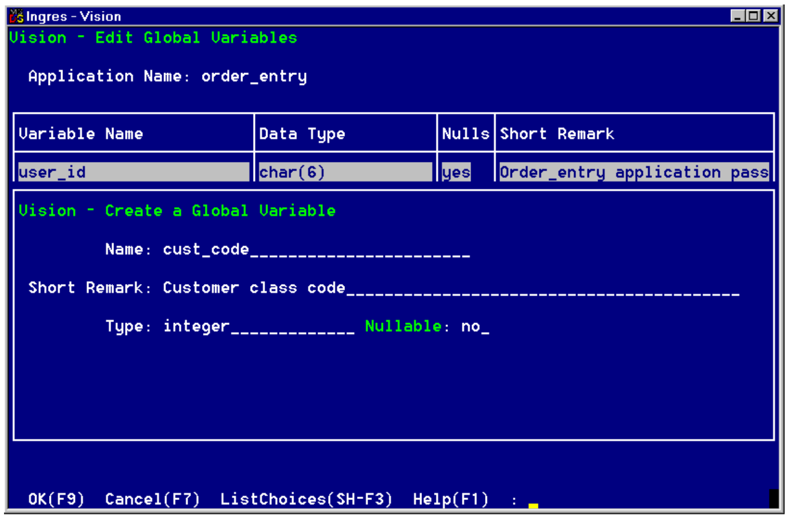
   Vision displays the Edit Global Variables window as shown in the following figure:



```
Ingres - Vision
Vision - Edit Global Variables

  Application Name: order_entry

 Variable Name            Data Type          Nulls Short Remark
 user_id                  char(6)            yes   Order_entry application pass

            Place cursor on row and select desired operation from menu.

  Create(SH-F1)  Destroy(SH-F2)  Edit(SH-F3)  Rename(SH-F4)  > :
```

4. Select Create from the menu.

   Vision displays the Create a Global Variable pop-up window as shown in the following figure:



```
Ingres - Vision
Vision - Edit Global Variables

  Application Name: order_entry

 Variable Name            Data Type          Nulls Short Remark
 user_id                  char(6)            yes   Order_entry application pass
 Vision - Create a Global Variable

         Name: cust_code_______________________

  Short Remark: Customer class code_________________________________________

         Type: integer_____________ Nullable: no_

  OK(F9)  Cancel(F7)  ListChoices(SH-F3)  Help(F1)  :
```

5. Enter a name for the variable.

6. Press Tab.

7. Enter an optional descriptive Short Remark.

   This remark appears in the Edit Global Variables window. It also appears as a comment in the 4GL code generated for the variable.

8. Press Tab.

9. Specify a data type in either of these ways:
   - Enter a valid Ingres data type or the name of a record type that you have created previously for this application. If the type requires a length, enter the length in parentheses.
   - Use the ListChoices operation.

     Vision displays a list of valid Ingres data types. If the type you select requires a length, Vision prompts you for the length. Enter the length without parentheses in the pop-up field. Vision places the data type in the Type field with the length, if required, in parentheses.

     For example, to indicate a char(20) data type, select "char," then enter "20" in the length popup. Vision displays the type as char(20).

     Note that you cannot use ListChoices for a record type.

   Be sure the data type is consistent with the function of the constant. For example, if the value of the variable is a percentage, use a float data type.

10. Press Tab to move the cursor to the next field.

    If you have specified a record type in the Type field, the Array field is displayed. Enter yes to indicate the variable is an array or no to indicate the variable is a single record.

    If you have specified an Ingres data type in the Type field, the Nullable field is displayed. The default value of yes indicates that the variable can accept a null value. Enter no to make the variable not nullable.

11. Select OK.

    Vision displays your global variable definition in the Edit Global Variables window.

12. Select End to return to the window in which you were working.

Assign an Initial Value to a Global Variable

By default, Vision sets the initial value of global variables to a blank string for character data types and zero for numeric data types.

To assign a different initial value to a global variable

1. Write a 4GL assignment statement.
2. Place this statement as "Form-Entry" escape code for the starting frame of the application (see How Writing Escape Code Works (see page 210)).

In this way, Vision compiles the code at the beginning of the application. Be sure to include this frame when you run the application. Otherwise, Vision does not see the assignment statement and uses the default values (a blank string or zero) instead.

Edit a Global Variable Definition

To edit the definition of a global variable to change its data type or nullability

1. Follow Steps 1 through 3 described above for defining a global variable.

   Vision displays the Edit Global Variables window.

2. Position the cursor on the name of the variable you want to change.

3. Select Edit.

   Vision displays the Edit a Global Variable Definition window. This window contains the information you entered when you created the variable (or the last time you edited it).

4. To change the data type, type the new data type over the current value and press Tab.

5. To change the Nullable or Array field, type yes or no as appropriate over the current value and press Tab.

6. Edit the descriptive Short Remark.

7. Select End.

   Vision displays the new definition in the Edit Global Variables window.

8. Select End again to return to the window in which you were working.

Rename a Global Variable

When changing the name of a global variable you have defined, be sure to change the name at each place you use the variable throughout the application, including any 4GL assignment statement or other escape code.

To rename a global variable

1. Follow Steps 1 through 3 as described above for defining a global variable.

   Vision displays the Edit Global Variables window.

2. Position the cursor on the name of the variable you want to rename.

3. Select Rename from the menu.

   Vision prompts you for the new name.

4. Enter a new name for the variable.

5. Press Return.

   The new name of the variable appears in the Edit Global Variables window.

Destroy a Global Variable

When destroying a variable that you no longer are using in an application, be sure to delete all references to the variable in the application, including any escape code, or Vision cannot compile and run your application.

To destroy a global variable

1. Follow Steps 1 through 3 as described above for defining a global variable.

   Vision displays the Edit Global Variables window.

2. Position the cursor on the name of the variable you want to destroy.
3. Select Destroy from the menu.

   Vision asks you to confirm that you want to destroy the variable.

4. Enter y and press Return.

   Vision destroys the variable and removes it from the Edit Global Variables window display.

Using Record Types

The data you use in Vision applications generally is in the form of standard Ingres character, numeric, date or money data types. However, you also can combine these data types to create your own data types, called *record types*.

A record type combines related components of data into a single value. Using a record type increases the efficiency and performance of your application by letting you retrieve, store, or pass multiple data items as a single item.

After you define a record type, you can use it in your application in various ways:

- In a local or global variable definition
- In a global constant definition
- In escape code
- As a parameter passed between frames

For example, your application might use a customer table that has four columns—street, city, state and zip code—to store addresses. You can define a record type that combines these four values into a single value. Retrieve the combined customer address into a single local variable and pass its value as a parameter to another frame.

Define a Record Type

To define a record type

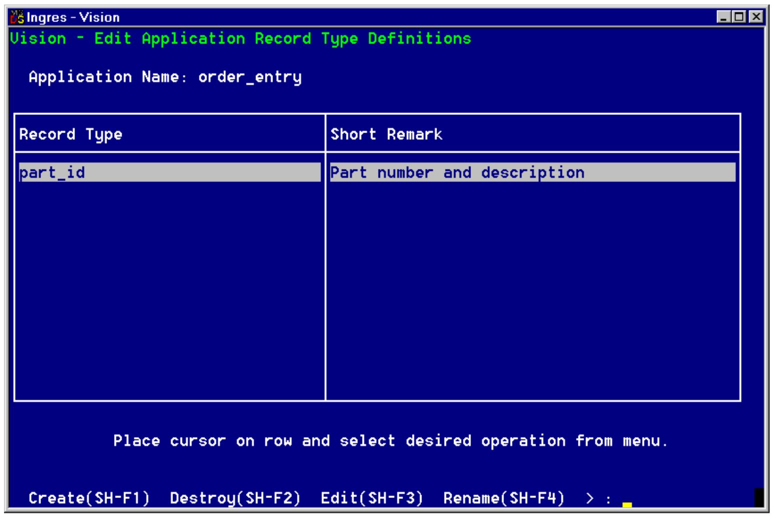
1. Select Edit from the menu in the Application Flow Diagram Editor or Visual Query Editor.

   Because the record type you define is global to the entire application, you can select any frame as the current frame.

2. Select Global Components from the list of options.

3. Select Record Types from the list of global components.

   Vision displays the Edit Application Record Type Definitions window as shown in the following figure:



```
Vision - Edit Application Record Type Definitions

  Application Name: order_entry

 Record Type                       Short Remark
 part_id                           Part number and description

            Place cursor on row and select desired operation from menu.

  Create(SH-F1)  Destroy(SH-F2)  Edit(SH-F3)  Rename(SH-F4)  > :
```

4. Select Create from the menu.

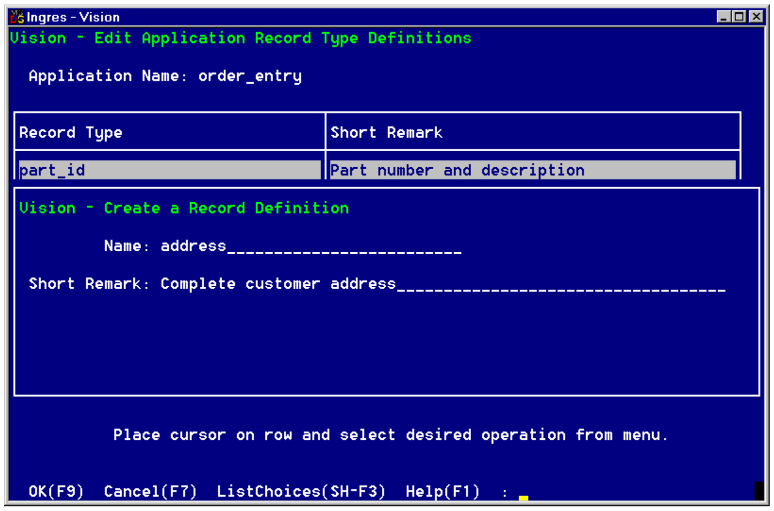
   Vision displays the Create a Record Definition pop-up window as shown in the following figure.

5. Enter a name for the record type.

6. Press Tab.

7. Enter a Short Remark to describe the record type.

   This remark appears in the Edit Application Record Type Definitions window. It also appears as a comment in the code that Vision generates for the record type.

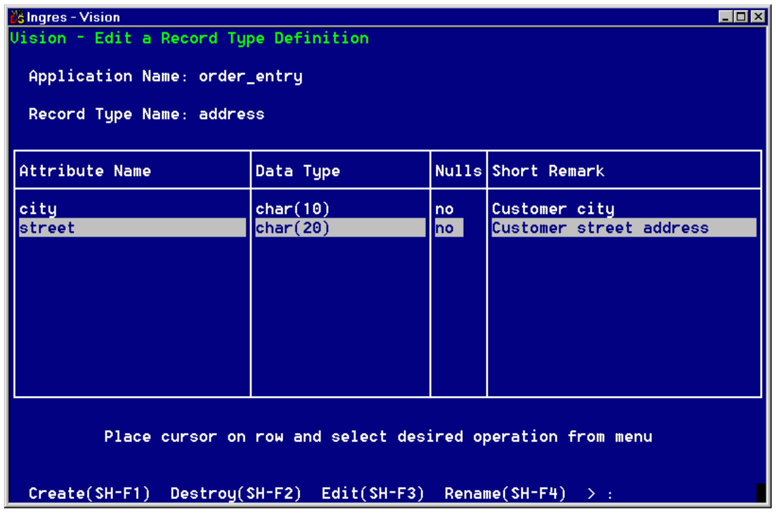


```
Ingres - Vision
Vision - Edit Application Record Type Definitions

  Application Name: order_entry

 Record Type                     Short Remark
 part_id                         Part number and description

 Vision - Create a Record Definition

         Name: address_________________________

  Short Remark: Complete customer address___________________________________

           Place cursor on row and select desired operation from menu.

  OK(F9)  Cancel(F7)  ListChoices(SH-F3)  Help(F1)  :
```

8. Select OK from the menu.

   Vision displays the Edit a Record Type Definition Window as shown in the following figure. You use this window to specify the various items, or *attributes*, that make up your record type.

   For example, "street" and "city" are two of the four attributes of the "address" record type in the example mentioned above.



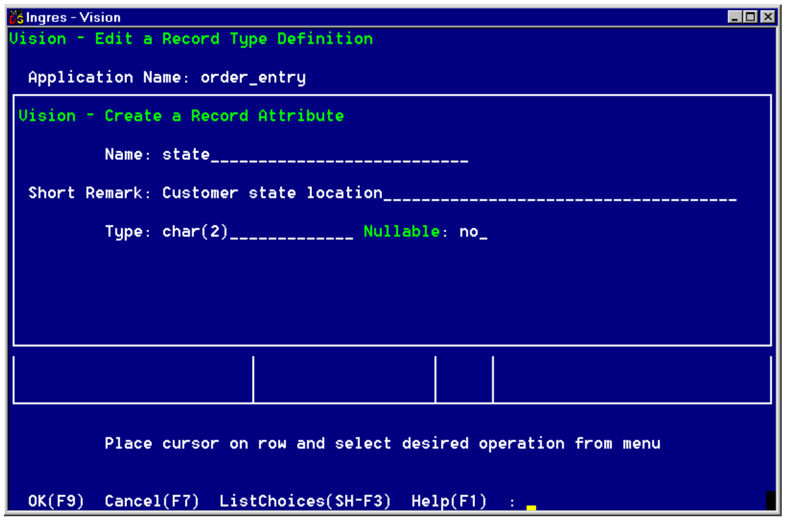
```
Ingres - Vision
Vision - Edit a Record Type Definition

  Application Name: order_entry

  Record Type Name: address

 Attribute Name           Data Type          Nulls Short Remark
 city                     char(10)           no    Customer city
 street                   char(20)           no    Customer street address

          Place cursor on row and select desired operation from menu

  Create(SH-F1)  Destroy(SH-F2)  Edit(SH-F3)  Rename(SH-F4)  > :
```

9. Select Create from the menu.

   Vision displays the Create a Record Attribute pop-up window as shown in the following figure.

10. Enter the name of an attribute and press Tab.
11. Enter a Short Remark to describe this attribute.
12. Specify a data type for the attribute in either of the following ways:
    - Enter the name of a standard Ingres data type or of another record type that you previously defined.
    - Use the ListChoices operation.
13. Press Tab.

    The cursor moves to the Nullable field. The default value of "yes" indicates that this component can have a null value.
14. Press Tab to accept the default or type no and press to make this component not nullable.



```
Ingres - Vision
Vision - Edit a Record Type Definition

  Application Name: order_entry

 Vision - Create a Record Attribute

         Name: state___________________________

  Short Remark: Customer state location_____________________________________

         Type: char(2)_____________ Nullable: no_



         Place cursor on row and select desired operation from menu

  OK(F9)  Cancel(F7)  ListChoices(SH-F3)  Help(F1)  :
```

15. Select OK from the menu.

    Vision places the attribute definition in the Edit a Record Type Definition window.
16. Repeat Steps 9 through 15 for each attribute of the record type.
17. When you have specified all the attributes for this record type, select End.

Edit a Record Type Definition

You can change the definition of a record type to:

- Include additional attributes
- Delete attributes
- Modify a current attribute to change its data type or nullability
- Rename a current attribute

To modify a record type definition

1. Follow Steps 1 through 3 in Define a Record Type.

   Vision displays the Edit Application Record Type Definitions window.

2. Position the cursor on the name of the record type.
3. Select Edit from the menu.

   Vision displays the Edit a Record Type Definition window.

4. Select the appropriate operation as follows:
   - To delete an attribute, select Destroy from the menu.
   - To rename an attribute, select Rename from the menu.
   - To add new attributes, select Create and follow Steps 10 through 15 in the previous section for defining a record type.
   - To modify an attribute, select Edit from the menu and enter new values as desired.
5. When finished, select End to return to the Edit Application Record Type Definitions window.

Rename a Record Type

When changing the name of a record type that you have defined, you also must change the name at each place you use the record type throughout the application.

To rename a record type

1. Follow Steps 1 through 3 in Define a Record Type.

   Vision displays the Edit Application Record Type Definitions window.

2. Position the cursor on the name of the record type.
3. Select Rename from the menu.
4. When prompted, enter a new name and press Return.

   Vision renames the record type.

Destroy a Record Type

When destroying a record type that you have defined, be sure to remove all references to the record type throughout the application; otherwise Vision generates an error when you compile and run the application.

To destroy a record type

1. Follow Steps 1 through 3 in Define a Record Type.

   Vision displays the Edit Application Record Type Definitions window.
2. Position the cursor on the name of the record type.
3. Select Destroy from the menu.

   Vision asks you to confirm that this is the record type you want to destroy.
4. Enter yes and press Return.

Vision destroys the record type and removes it from the list of record type.

How Writing Escape Code Works

Vision generates the 4GL code for Menu, Append, Browse and Update frames based on the Application Flow Diagram and visual query specifications. You can also add functions to these frames not provided by the Vision-generated code by writing 4GL statements called *escape code*. Vision generates the basic code for your application and includes escape code to perform additional functions that are not part of the visual query.

For example, you can write escape code to:

- Display messages or prompts when the user calls a frame, enters a field, or selects a menu item
- Keep track of how many records are retrieved, updated, or appended on a frame
- Set "if-then" conditions for calling frames or taking other actions
- Assign initial values to global or local variables

You can use any 4GL statements in your escape code, subject to the restrictions discussed below. Your escape code must follow the standard 4GL syntax, with a semicolon (;) at the end of each statement.

The 4GL reference part of this guide discusses how to write 4GL code and provides a complete description of each statement and its syntax.

When you write escape code for a frame, you must specify its type, as determined by the point at which the escape code is activated when Vision runs the frame. For example, Form-Start escape code is activated before a form is displayed.

In addition to the escape code itself, Vision also generates a comment in the source code file so that you can locate the escape code. These comments are especially helpful when you are trying to correct errors in the code you wrote.

As a simple example of escape code, you can have Vision generate a message when a user starts the application. To do this, include a 4GL message statement on the top frame.

The escape code looks like this in the source code file:

```
/*# BEGIN Form-Start */
        message 'Welcome to the Order Entry System';
/*# END Form-Start */
```

When a user starts the application, the message appears before the form is displayed for the top Menu frame.

Additional examples of escape code are given later in this section.

Types of Escape Code

An escape code's type indicates the point at which the code is executed when the application is run. Certain types of escape code can be used only with specific frame types, as indicated in the following table.

The following table describes the types of escape code that are available in Vision:

Escape Type	When Executed	Frame Types
Form-Start	When the form is displayed	Menu, Append, Browse, Update
Form-End	After the user selects End (and provides confirmation when required), but before exiting a form	Menu, Append, Browse, Update
Query-Start	Before a query is run, just before the "Selecting	Browse, Update

Escape Type	When Executed	Frame Types
	Data..." message appears in the window	
Query-New-Data	After new data (Master records and any matching Detail records) are retrieved from the database, but before the user sees the data	Browse, Update Cannot use this escape type on a frame whose Master table is displayed as a table field.
Query-End	After all records (or no records) are returned by the query, and the user has exited the Go submenu	Browse, Update
Append-Start	After a user enters data on the form, before data is inserted in database	Append and Update (if appends are allowed)
Append-End	After a user enters data on the form, after data is inserted in database	Append and Update (if appends are allowed)
Update-Start	After a user enters data on the form, before the database is updated	Update
Update-End	After a user enters data on the form, after the database is updated	Update
Delete-Start	After a user enters data on the form, before data is deleted from database	Update (if deletes are allowed)
Delete-End	After a user enters data on the form, after data is deleted from database	Update (if deletes are allowed)
Menu-Start	After the user selects a specific menu item, before the called frame is displayed	Menu, Append, Browse, Update
Menu-End	After returning from the frame called by the specified menu item	Menu, Append, Browse, Update
Before-Field-Entry	When the user enters a form field	Menu, Append, Browse, Update
After-Field-	When the user exits from	Menu, Append, Update

Escape Type	When Executed	Frame Types
Change	a field, if the user typed a value into the field. The code is not activated if the value of the field is changed in another way, such as selecting a value using the ListChoices menu item.	
After-Field-Exit	When the user exits from a field; if the field also has Field-Change trigger code, the Field-Change code executes first	Menu, Append, Update
TableField-Menuitem	When the user selects the menu item from the menu	Menu frames with table-field menus
Menuline-Menuitem	When the user selects the menu item from the menu	Menu, Append, Browse, Update
Before-Lookup	When the user selects ListChoices, or another action which calls a lookup table, before the lookup table is displayed	Append, Browse, or Update frames which have lookup tables in their visual query
After-Lookup	When the user selects ListChoices, or another action which calls a lookup table, after the lookup table is displayed and before returning to the calling frame	Append, Browse, or Update frames which have lookup tables in their visual query
On-Timeout	When the user has been idle for the specified timeout period. The default period is 300 seconds. You can change the timeout period in the template files, or by using escape code. For example, in Form-Start escape code: set forms frs (timeout=200)	Menu, Append, Browse, Update

Escape Type	When Executed	Frame Types
On-Dbevent	When a specified dbevent occurs. The event must be a registered dbevent. You must register the event in the application by including a register dbevent statement, usually through Form-Start escape code. For more information on dbevents, see the 4GL reference part of this guide.	Menu, Append, Browse, Update

Guidelines for Including Escape Code

You can include most 4GL statements in your escape code. However, be aware of these restrictions and implications:

- Do not use begin-end loops that span escapes.

  For example, do not put a begin statement into a Field-Entry escape and the corresponding end statement into a Field-Exit escape.

- Do not include a set autocommit on statement.
- Do not issue an initialize statement in escape code for a frame, because the Vision code generator always creates this statement for the frame.
- Do not use activation blocks—such as Menu item or Field activations—in escape code.

  However, you can use submenus and queries with attached menus.

- Any escape code that you indicate for a particular menu item or field supercedes escape code specified with the ALL option (described below) for all menu items or fields of a frame.
- If you write escape code that calls another frame, the called frame does not appear on the Application Flow Diagram.
- If you use database (update, delete, select, and insert) statements in your escape code, include error checking code to ensure the data in your database is being updated correctly.

Specifying Field or Menu Item Escape Code

Certain escape code types refer to a field or menu item. Field escape codes are: Before-Field-Entry, After-Field-Change, and After-Field-Exit. Menu item escape codes are: Menu-Start and Menu-End.

When you create field or menu item escape code, Vision displays a selection list for you to specify to which field or menu item the code refers.

- For field escape code, the selection list includes all fields displayed on the form, including columns from the Master and Detail tables, and any fields that you have created on the form.

  The columns of a table field are preceded by "iitf." If there is a Detail table, the Detail table is a table field. For example, the part_no column of a Detail table appears as iitf.part_no. If there is no Detail table, a Master table can be displayed as a table field.

  For Before-Field-Entry and After-Field-Exit escape types, the selection list also includes an "ALL" option to activate the escape on all fields of the Master table (if simple fields) and an "iitf.ALL" option to activate the escape on all table-field columns of the Detail table (or the Master table if it is a table field).

- For menu item escape code, the selection list contains the menu items that you generated through the Application Flow Diagram. That is, the menu items associated with the child frames.

Using Resume Statements in Escape Code

When you use a resume (or resume field, resume next, resume nextfield, or resume previousfield) statement in your escape code, control of the application leaves the escape code block and returns to the 4GL source code that called the escape code. Any 4GL statements that follow the resume statement in the source code file are skipped when the application is running.

The escape types listed below are the only ones that are not followed by Vision-generated 4GL statements in the source code files for the frames in which they are included:

- Query-New-Data
- Before-Field-Entry
- After-Field-Exit
- Menuline-Menuitem

If you use a resume statement with other escape code types to skip 4GL statements deliberately, check the generated code to make sure control occurs as desired. If not using a resume statement, position the cursor on a particular field simply by editing the form and changing the order of the fields.

Purging Unneeded Escape Code

If you delete a form field or menu item for which you have included escape code, Vision purges the escape code that refers to the deleted item. The escape code is purged from the frame's source code file the next time Vision regenerates the code.

You can force Vision to generate the new source code by using the Compile operation on the Application Flow Diagram Editor menu.

Add and Delete Menu Operations

Add menu operations to single-line or table-field menus, or delete menu operations from table-field menus by using Menuline-Menuitem escape code to add menu operations to the single-line menu at the bottom of the window. You cannot use escape code to delete menu operations from a single-line menu.

In a Vision table-field Menu frame, you can use escape code to add or delete menu operations.

To add or delete menu operations

1. Use Form-Start escape code to add or delete rows to the menu's table field before it is displayed. For details, see How You Can Change the Menu Item Text in the Table Field.
2. If you are adding menu operations, use TableField-Menuitem escape code to specify the action that occurs when the menu item is selected. For details, see the Coding a Menu Operation for a Table-Field Menu Frame section.
3. You can also edit the form to change the number of table-field rows displayed in the window. Otherwise, Vision creates the form based on the initial number of menuitems in the table (the number of called frames). For example, if you add one menu item to a table field with five called frames, the form created by Vision has five rows. You must scroll the table field to view the new sixth menu item. To display all six menu items, edit the form to display six rows.

How You Can Change the Menu Item Text in the Table Field

When a user selects a table-field Menu frame, Vision loads a table field with the menu item text for each child frame. Each row contains a Command field and an Explanation field. Use Form-Start escape code to add or delete rows in this table field before it is displayed.

For example, to display a message, use the following Form-Start escape code:

```
loadtable iitf
        (command = 'Message', explanation = 'Display message');
```

For example, to remove the first menu item in the table, use the following Form-Start escape code:

```
deleterow iitf[1]
```

How You Can Code a Menu Operation for a Table-Field Menu Frame

Use Tablefield-Menuitem escape code to tell Vision what to do when the user selects a menu item that you have added in the Form-Start code.

Vision uses two 4GL local variables to handle menu operations:

- iichoice represents the current row in the table field at the time the user chooses Select
- iifound is an integer variable whose value is determined as follows:
- If the user selects a menu item to call a child frame, then Vision sets iifound to 1 (true).
- If the user selects a menu item for an operation that you have added, then Vision sets iifound to 0 (false).

    If iifound is 0, Vision displays an error message that the command is not found.

Your Tablefield-Menuitem escape code must indicate what to do when the user selects a menu item that you have added in the Form-Start code (that is, when iifound is 0). If you have added more than one menu item, your Tablefield-Menuitem code must use the value of iichoice to determine which operation to execute.

Be sure to reset iifound to 1 at the end of the Tablefield-Menuitem code. Otherwise, Vision gives an error message.

For example:

```
if iichoice = 'message' then
        message 'you have chosen the ''message'' command.'
        with style = popup;
        iifound = 1;
 endif
```

View Examples of Escape Code

This section provides examples of various types of escape code that you can use in developing a Vision application. You also can view additional examples on your terminal display while you are creating escape code in Vision.

To see examples of escape code while you are using Vision

1. Select Edit from the menu in the Application Flow Diagram or visual query display window for the current frame.
2. Select Escape Code from the list of options.
3. Select Help from the menu.

   Vision displays information about escape code types.
4. Select SubTopics from the menu.

   Vision displays a selection list of escape code types.
5. Select the escape code type for which you want to see examples.

   Vision displays examples of this escape code, as well as any additional warnings or restrictions about using this type.

Alternatively, you can display the examples after you have selected a type and displayed the blank window for entering the actual escape code. To do this, select Help when Vision displays the window in which you enter your escape code, as described in the procedures below.

You also can use the SubTopics operation of the Help facility to display examples of standard 4GL statements if you need help with syntax while entering your escape code.

Escape Code Examples

The following are examples of escape code to perform various functions in an application.

- Form-Start escape code to check the password that a user enters against a table of authorized users and their passwords:

  ```
  pswd = prompt noecho 'Enter your password: '
  with style = popup;

  formname = select cnt = count(*)
  from users
  where user = dbmsinfo ('username')
   and password = :pswd;

  if (cnt = 0) then
    message 'No authorization for this application'
      with style = popup;
      exit;
   endif;
  ```

- Form-Start escape code to retrieve data into local variables:

  ```
  formname = select h_name= e.name,
   h_status   = e.status
  from employees e
  where e.empno = :empno;
  ```

- Menu-Start escape code to verify that a particular field contains a value before calling the frame. This example shows using a resume statement to skip Vision-generated code:

  ```
  if (last_name = '') then
      callproc beep(); /* a 4gl procedure */
      message 'You must enter a last name'
      with style = popup;
   resume;
  endif;
  ```

- Menu-End escape code to return a value based on what action the user takes on a frame. This example uses a built-in global variable called "IIretval" to communicate between frames:

  ```
  if (IIretval = 2) then
      /* user selected TopFrame in called frame */
  elseif (IIretval = -1) then
      /* an error occurred in called frame*/
  else
      /* called frame finished or user selected end */
  endif;
  ```

- Query-Start escape code to retrieve data into table fields on a form:

  ```
  formname.tblfldname = select name = c.name,
       balance = c.balance
  from customers c
  where c.custno = :custno;
  ```

- Query-End escape code to check how many rows the user selected:

  ```
  inquire_sql (rows = rowcount);
  ```

- Append-Start escape code on an Update frame to increment the sequenced field "seq_key" in the "orders" Master table:

  ```
  seq_key = callproc sequence_value
      (table_name =   'orders',column_name = 'order_no',
      increment = 1, start_value= 1);
   commit work;
  ```

  For more information on specifying sequenced fields, see Defining Append frames in Defining Frames with Visual Queries (see page 117).

- Form-Start escape code to pass a last name as a parameter to a Browse or Update frame. The child frame:
  - Has the Master table displayed as a table field
  - Has the Qualification Processing frame behavior disabled

  The value is passed into a local variable (called "l_lname") on the child frame. The escape code then assigns the passed value into the table field for the Master table, where the value is used to qualify the query that determines the first record that a user sees.

  ```
  /* form-start escape code for child frame */

  insertrow iitf[0]: /*open a row in the table field*/

  /* Assign passed-in value to newly-opened row. Then
  use this value to qualify the query.  */

  if (l_lname != '') then /* A value was passed */
      iitf[1].lname = l_lname;
   endif;
  ```

- After-Field-Exit escape code to update a column in the database with the result of the date_trunc function. If the date is entered incorrectly, the function returns NULL. To ensure that the data in the database is not overwritten with a NULL value, include error checking in your escape code.

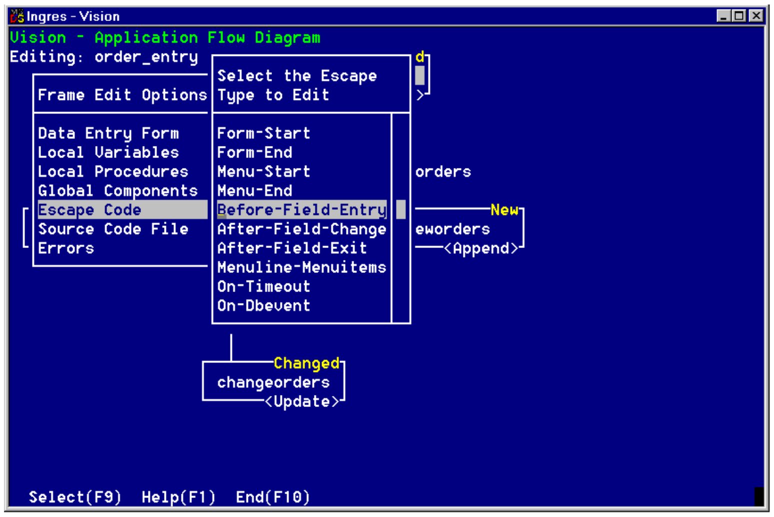
  ```
  mo_date := date_trunc( 'day', mo_date );
   inquire_forms frs ( IIerrorno = errorno );
   if IIerrorno != 0 then
      message 'Invalid date format entered - Please '
              + 'REENTER' with style = popup;
   resume; /* Breakout of current display loop and go
      back to the field w/ bad data */
  endif;
  ```

Create Escape Code

To create escape code for a frame

1. Select Edit from the menu in the Application Flow Diagram or visual query display window for the current frame.
2. Select Escape Code from the list of options.

   Vision displays the types of escape code available for this frame type. An asterisk (*) appears after any type for which this frame already has escape code (see the following figure):

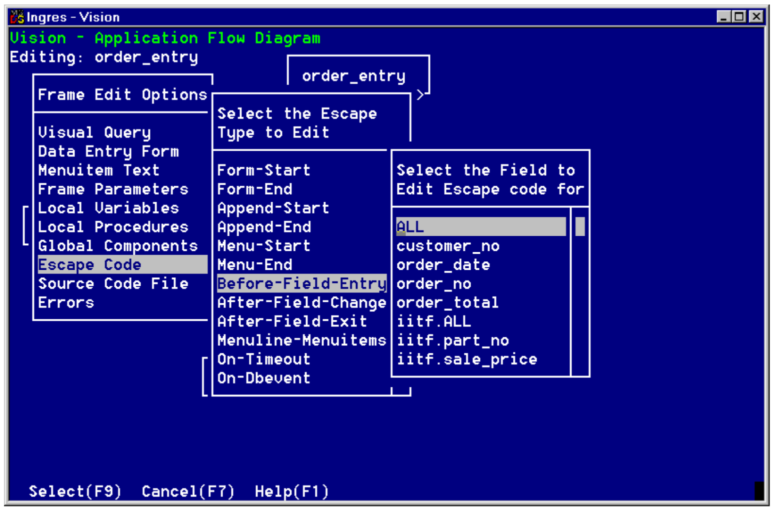


```
Ingres - Vision
Vision - Application Flow Diagram
Editing: order_entry

Frame Edit Options   Select the Escape
                     Type to Edit

Data Entry Form      Form-Start
Local Variables      Form-End
Local Procedures     Menu-Start              orders
Global Components    Menu-End
Escape Code          Before-Field-Entry      New
Source Code File     After-Field-Change      eworders
Errors               After-Field-Exit        <Append>
                     Menuline-Menuitems
                     On-Timeout
                     On-Dbevent

                          Changed
                     changeorders
                         <Update>

 Select(F9)  Help(F1)  End(F10)
```

3. Select an escape code type.

   For all types except those related to fields and menu items, Vision displays a blank window in which you can enter your escape code (see the figure under Step 4 below). Proceed to Step 5.

   For escape code related to a field or menu item, Vision displays a selection list of fields or menu items as shown in the following figure:

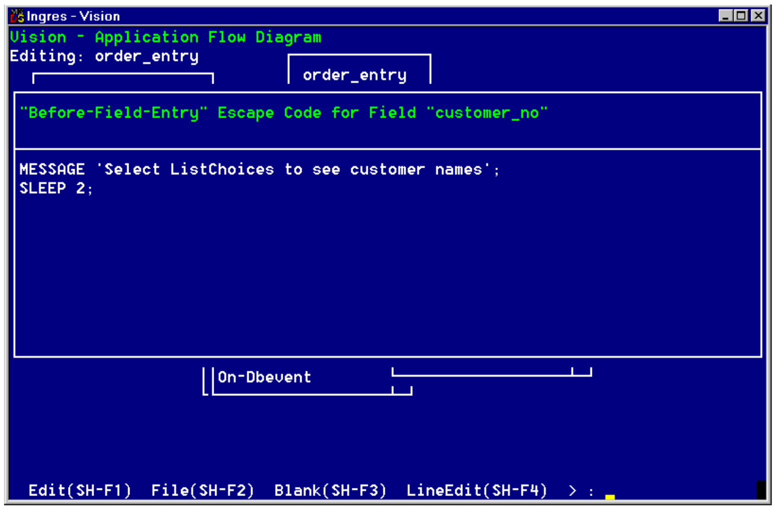


```
Ingres - Vision
Vision - Application Flow Diagram
Editing: order_entry
                                        order_entry
  Frame Edit Options
                           Select the Escape
  Visual Query             Type to Edit
  Data Entry Form
  Menuitem Text            Form-Start           Select the Field to
  Frame Parameters         Form-End             Edit Escape code for
  Local Variables          Append-Start
  Local Procedures         Append-End           ALL
  Global Components        Menu-Start           customer_no
  Escape Code              Menu-End             order_date
  Source Code File         Before-Field-Entry   order_no
  Errors                   After-Field-Change   order_total
                           After-Field-Exit     iitf.ALL
                           Menuline-Menuitems   iitf.part_no
                           On-Timeout           iitf.sale_price
                           On-Dbevent

 Select(F9)  Cancel(F7)  Help(F1)
```

4. Select the name of a menu item or field (or the appropriate ALL option).

   See the Specifying Field or Menu Item Escape Code section for more information.

   Vision displays a pop-up window in which you can enter your escape code as shown in the following figure:



```
Ingres - Vision
Vision - Application Flow Diagram
Editing: order_entry
                                        order_entry
 "Before-Field-Entry" Escape Code for Field "customer_no"

 MESSAGE 'Select ListChoices to see customer names';
 SLEEP 2;

                           On-Dbevent

 Edit(SH-F1)  File(SH-F2)  Blank(SH-F3)  LineEdit(SH-F4)  > :
```

5. Enter your 4GL escape code in the pop-up window.

   You can enter your escape code in any of the ways described in the Entering Escape Code section. This section also describes the menu items available from the pop-up window.

6. Select Save to save your code.

7. When finished, select End.

   Vision redisplays the selection list of escape code types. You now can do either of the following:

   - Repeat Steps 3 through 7 to create or edit other types of escape code for this frame.
   - Select End to return to the window in which you were working.

Entering Escape Code

There are several ways in which you can enter escape code:

- Directly in the escape code pop-up window
- With the default system editor
- From an external file

Each of these methods is described below.

How You Can Enter Escape Code Directly in the Window

When Vision displays the pop-up window for writing escape code, you can enter your code directly in the window. The standard Ingres window editing keys and functions are available.

The following edit operations are available in the escape code pop-up window:

Edit

Calls the editor for you to write your escape code, as described in How You Can Enter Escape Code Using the System Editor, in this topic

Blank

Clears the window. Be aware that there is no "undelete" function to restore text after you have cleared the window.

LineEdit

Displays a submenu with these line editing functions:

InsertLine

Inserts a blank line above the line on which the cursor is positioned

DeleteLine

Deletes the line on which the cursor is positioned

SplitLine

Divides a line into two lines at the point where the cursor is positioned

JoinLines

Moves the next line to the end of the line on which the cursor is positioned

Variables

Lets you create or edit local variables or global components to use in escape code

Save

Saves the escape code you have entered

How You Can Enter Escape Code Using the System Editor

You can use your system editor to write your escape code, rather than using the line edit functions. Vision uses the default system editor.

See your Ingres system administrator if you want to change this default.

To write escape code using your system editor:

1. When Vision displays the pop-up window for entering escape code, select Edit from the menu.

   Vision calls the system editor and opens a file.

2. Enter your code in the window, using the standard editor operations.
3. When you are finished entering your code, save your file and exit from the editor.

   Vision places the text from the file into the escape code pop-up window.

To edit the file containing this code, select Edit again. Vision redisplays the file.

How You Can Enter Escape Code Using External Files

When editing escape code, you can use external text files to:

- Write text from the window to a file.
- Read text from a file to the window.

Using an external text file is especially convenient if you are using the same or similar escape code in more than one place. You can write the code to a file when you first create it, then read in this file for additional instances of escape code.

When you write escape code to an external file, you simply are using it to store your code. Vision does not access the file itself when it runs a frame containing the escape code.

You can store a basic version of the code in the file, call the file, and change the code as needed each time you use it.

To write escape code to a file

1. Enter the code in the escape code pop-up window.
2. Select File from the menu.
3. Select WriteFile from the submenu.

   Vision prompts you for a file name.
4. Enter the name of the file in which you want to save your escape code.

   If the file is not in the current directory, be sure to enter the entire directory path.
5. Press Return.

   Vision saves your text in the file and returns you to the escape code pop-up window.

To read escape code from a file

1. Display the pop-up window for entering escape code.
2. Select File from the menu.

3. Select ReadFile from the submenu.

   Vision prompts you for the name of the file.

   If the file is not in the current directory, be sure to enter the entire directory path.

4. Enter the name of the file containing the escape code and press Return.

   Vision writes the file to the window. If any text is in the window, Vision places the contents of the file above the line on which the cursor is positioned.

If you make any changes to the file text while it is displayed in the window, you can either write the new text back to the file or create a new file.

Revise Escape Code

To modify escape code

1. Follow Steps 1 through 4 for creating escape code in Create Escape Code (see page 221). Be sure to select the correct frame and escape code type (and fieldname or menu item, if applicable) to display the code you want to edit.

   Vision helps you by placing an asterisk (*) next to each frame type in the selection list for which escape code already exists.

2. Edit the code as described in Entering Escape Code (see page 223).

3. Select End to save your changes.

Verifying Escape Code

Vision does not detect any errors in your escape code until it compiles it with the code it generates for the frame. To check for code errors, you must test the frame.

Correct your errors in either of these ways:

- Use the Vision error handling facility.
- Edit your escape code as described above, and then test the frame again.

For information on how to test frames and use Vision error handling facility, see Completing a Vision Application (see page 259).

Chapter 9: Modifying Vision Code

This section contains the following topics:

Vision creates the 4GL source code and forms for the Vision frames in your application. As you build your application, you can make changes through the Visual Query Editor and the Application Flow Diagram. Vision regenerates the code to incorporate these changes. However, sometimes the code generated by Vision does not provide the desired behavior. This chapter discusses how to add to or override the code generated by Vision through:

- Writing escape code
- Modifying the template files
- Including source code processing commands
- Editing the source code to create a Custom frame
- Editing help files

Note: If you edit the source code, you lose the advantages of using the Vision code generator. Therefore, we recommend that you try one of the other methods before editing the source code.

The following section summarizes the first four methods. The sections after that discuss all five methods in detail.

Summary of Methods to Modify Code

The following table describes the methods to modify code:

Method	Marks Frame as "Custom"	Required Knowledge	Range	Advantages/ Disadvantages
Writing Escape Code	No	4GL	Affects individual frames	Easy and direct.
Modifying	No	4GL and	All frames	A good way of making global changes to

Method	Marks Frame as "Custom"	Required Knowledge	Range	Advantages/ Disadvantages
Template Files		template language	that use that template file. Can be used to modify all frames of a particular type.	an application. More difficult than writing escape code.
Including Source Code Processing Commands	No	Depends on how implemented. Often requires extensive knowledge of operating system or a 3GL	Individual or all frames	Can require considerably more work to implement than the other methods. Edits your source code after it is generated by Vision but the result is not considered "Custom" code. Can require alterations to work with subsequent Vision releases.
Editing Source Code	Yes	4GL	Individual frames	Very quick and easy. Code is considered "customized" by Vision; any code changes must be reapplied.

Writing Escape Code

Writing escape code is a way to incorporate 4GL statements into your application. You can add escape code to various places in your application. The 4GL statements that you write are included as written into the 4GL code that Vision generates for the code. For example, Menu-Start escape code is added to the source code file after the menu item but before the generated code for that menu item.

Note: We recommend that you use escape code rather than editing the source code directly. By using escape code, you can still take advantage of the Vision code generator for incorporating changes to your forms, templates, visual queries, and so forth, without overwriting your customizations.

For example, if you want to change the menu items of a menu, you can accomplish this without editing the code directly in the source code file. Instead, write your menu item code as Menu-Start escape code. Include a resume statement to skip the Vision-generated menu choices. Because your menu item code is stored as escape code, you do not need to reapply the changes to each menu item if you change the visual query and need to regenerate the source code.

For more information on using escape code, see Using Vision Advanced Features (see page 177).

Modifying the Template Files

To create the source code, help files, and forms for the frames in your application, Vision combines your visual queries and other frame definition specifications with a set of frame templates and standard help files.

Modify a template file to change any Vision frame that uses that template by using one of the following:

- Change the text or the order of Vision-generated menu items
- Include additional menu items in your applications
- Edit messages that Vision displays to the user
- Modify the error recovery behavior that Vision performs after a query

Note: To modify a template file, make a copy of the template file in a new location, and modify the copy of the file. (See Template File Locations (see page 229) for directions.) Never make direct updates to the template files that you received with Vision.

Template File Locations

By default, Vision uses the template files in the following directory:

Windows: %II_SYSTEM%\ingres\files\english

UNIX: $II_SYSTEM/ingres/files/english

VMS: II_SYSTEM:[INGRES.FILES.ENGLISH]

Important! Never make changes directly to a template file in the above named directory.

Modify Template Files

To modify template files

1. Copy the template files from the default directory to another directory.
2. Define the Ingres logical or environment variable II_TFDIR to point to the new directory. For example:

   Windows: set II_TFDIR=\apps\orderapp\templates

   UNIX: setenv II_TFDIR /apps/orderapp/templates

   VMS: define II_TFDIR apps:[orderapp.templates]
3. Edit the template file copies in the new location. Modify the template files according to the grammar described in Template Files Reference (see page 319).

To use the modified versions of the template files, you must define II_TFDIR every time you use Vision, before you start Vision. If you want all users to use your modified template files, be sure II_TFDIR is set installation wide. You can define II_TFDIR in a startup file so that the installation always points to this directory.

If II_TFDIR is defined, each time Vision generates code for a frame, it first looks for a template file (including ##INCLUDE template files within the frame template) in the directory pointed to by II_TFDIR.

If Vision does not find a template file in the directory pointed to by II_TFDIR, it checks the default location. If it cannot find the template file in either place, an error occurs and code is not generated. (The previous version, if any, of the 4GL file is not affected.)

Types of Template Files

There is a separate template file for each frame type. The frame type template files include additional template files that provide information about menu items. There are also template files for Help files.

Template Files for Frames

Vision uses a template file to generate the 4GL code for each Vision frame type. These files are described in the following table:

File Name	Generated Frame Type
ntmenu.tf	Menu frames (no tables used)
msappend.tf	Append frame, Master table only (simple fields)

File Name	Generated Frame Type
mtappend.tf	Append frame, Master table only (table field)
mdappend.tf	Append frame, Master and Detail tables
msbrowse.tf	Browse frame, Master table only (simple fields)
mtbrowse.tf	Browse frame, Master table only (table field)
mdbrowse.tf	Browse frame, Master and Detail table
msupdate.tf	Update frame, Master table only (simple fields)
mtupdate.tf	Update frame, Master table only (table field)
mdupdate.tf	Update frame, Master and Detail table

Included Files

The frame templates refer to additional template files that are included as needed when the code for the frame is generated. These files provide additional specifications for a frame, such as Vision-generated menu items. The names of the included files:

- Begin with "in"
- Have a file extension of ".tf"

Included files can be used by more than one frame template.

Help File Templates

Vision also provides help file templates. Vision uses the help file templates to create one or more help files for each frame in the application. The Vision help files contain generic information about the frame type. You can change the template; however, it is often more useful to customize each frame's help file after it is generated. To change the generated files, see Editing Vision-generated Help Files (see page 242).

Vision generates the help file for each frame by:

- Making a copy of the appropriate template help files for the frame type

  The names of template files for generating Help windows all begin with "fg" and have the extension "hlp." Each ##generate help statement in the template file provides the name of the help file that is to be copied.
- Giving this file the same name as the frame and the extension ".hlp"

  Help files with the extension ".hla" and ".hlq" are generated for Update frames that contain submenus.
- Placing the file in the specified source code directory for the application

Components of Template Files

Template files are made up of several components. The components of a template file are:

Substitution variables

Global substitution variables that represent various components and specifications of a frame. Any string beginning with a "$" is read as a substitution variable. Substitution variables are represented here as $*variable*.

Logicals or environment variables

A variable set outside of Ingres. Any string beginning with "$$" is read as a logical or environment variable. They are represented here as $$*logical*.

"Template language" executable statements

Executable statements in a template language tell Vision how to generate the code. Lines beginning with "##" are template language executable statements.

The code generator converts these statements into 4GL source code; the "##" statements do not appear in the generated source code. For more information on template language statements, see Template Files Reference (see page 319).

4GL code

Code inserted as is into the generated code. 4GL code can include any standard 4GL statements.

The following sections describe how you can use these components to customize your template files. You can:

- Change or add 4GL code
- Change or add template language code, including defining new substitution variables

For a complete list of substitution variables and template language statements, see Template Files Reference (see page 319).

Format

The code generator determines the type of component by its format. Therefore, if you make any changes to a template file, you must observe the formatting rules, or Vision cannot generate the 4GL code.

The code generator keeps track of blanks and white space in a template language executable statement and reproduces those blanks and white space prior to any generated 4GL code for that statement.

Follow the existing layout to ensure that the generated code is properly indented and easy to read.

An Example of Template File Code

The following is a sample fragment of a template file for an Update frame with a Master and Detail table:

```
## IF $delete_master_allowed THEN
## INCLUDE inmsdlmn.tf --Delete menuitem
## ENDIF
```

The code generator interprets this code as follows:

- The "##" on each line indicates an executable template language statement
- The first line begins an executable if-then statement
- The substitution variable *$delete_master_allowed* is a boolean that returns TRUE if the frame definition lets users delete records from the Master table
- ## INCLUDE on the second line is an executable statement to include and process another template file if deletions are allowed
- "inmsdlmn.tf" is the name of the included template file that generates a Delete menu item
- The text "--Delete menuitem" is a template file comment to document the purpose of this line; this text does not appear in the generated code
- ##ENDIF indicates the end of the executable if-then statement

Using Substitution Variables in Template Files

Vision template files use global variables called substitution variables. These variables have the form *$variable*.

When Vision generates the code for a frame, actual values are substituted for *$variables*. Variable substitution takes place everywhere, even inside quoted strings and comments.

When substitution variables are used in executable "template language" statements—such as the **##** generate statement—that are translated into 4GL statements, the values are substituted before the statement is evaluated.

Substitution variables can be system-defined or user-defined:

- System-defined substitution variables represent the information about a frame that you specified with the Visual Query Editor or other Vision components. For example, the variable *$master_table_name* represents the name of the master table you specified when you created the frame.

  You cannot directly change the value of a system-defined substitution variable in a template file, but you can use them in your own 4GL or template language code in template files.
- User-defined substitution variables are variables that you create with the **##** define statement. Assign a value to the substitution variable when you create it and include the value in 4GL or template language code. Use the **##** undef statement to undefine a user-defined substitution variable.

User-defined and system-defined substitution variables are described in detail in Template Files Reference (see page 319). The tables in the appendix list the substitution variables recognized by the code generator.

Use the name of a logical or environment variable as the value of a substitution variable, as discussed in Using Logicals or Environment Variables in Template Files (see page 234).

Using Logicals or Environment Variables in Template Files

The Vision code generator treats the string $$*logical* in a template file as the name of a logical or environment variable. If *logical* has been defined, Vision substitutes its value into the generated 4GL code wherever the $$*logical* string appears in the template.

For example, if you have defined II_TFDIR to a specific directory, the code generator substitutes that directory wherever the string $$II_TFDIR appears in the template file.

Logicals in 4GL Statements

If $$*logical* appears in a 4GL statement (that is, a non-executable statement) in a template file, its definition is written directly into the generated 4GL source code.

The string $$*logical* is written directly into the generated code as it appears in the template file if:

- The logical or environment variable has not been defined
- The code generator cannot locate the value of *logical*

Include any number of logicals in 4GL statements in template files. However, logicals cannot be nested; that is, the definition of a logical is not checked to see it if refers to another logical. (An exception to this restriction is when the operating system performs the translation.)

Logicals in Executable Statements

If $$*logical* appears in an executable template language statement (one that begins with "##"), the code generator translates the logical before it evaluates the statement. If the logical has not been defined, an empty string is substituted for $$*logical*.

Include no more than one logical in each ## statement. If more than one appears, the code generator translates all the logicals to the definition of the last one in the statement.

Verifying That a Logical Is Defined

The most common way to include a logical is to use it within an ##ifdef/else/endif block. The code within the block is included in the generated source file only if the logical exists and has been defined to a value other than the empty string.

For example:

```
##IFDEF $$logical
        /* put this code in generated 4GL file if $$logical exists*/
##ELSE
        /* put this code in generated 4GL file if $$logical does not exist */
##ENDIF
```

Do not use $$*logical* variables in **##**if-endif statements.

Size Limitations for Logical Definitions

The definition of a $$*logical* in a template file must be no longer than 256 characters. (This limit is longer on some operating systems.) If the definition is longer than the allowed length, then only the allowed number of characters is included in the generated code.

Changing Template Language Statements

Template files include executable statements in a template language to tell Vision how to generate the code. The code generator executes the template language statements to produce the source code for a frame. These statements themselves do not appear in the generated 4GL code. Executable statements begin with "##".

The executable statements used by the code generator are listed in Template Files Reference (see page 319). You can add or delete statements to a template file, but you cannot change the basic format of a statement.

Changing 4GL Code in Template Files

The template files contain 4GL statements that Vision inserts directly into the source code files, for example, messages and text for generated menu items. Because the code is written directly into the source file, you must observe the usual 4GL syntax rules.

Examples of Altering Template Files

To alter the template files, use one of the following:

- Change the text of error messages or prompts. Use your text editor to change the wording of error messages in the template files.
- Add a menu item for a commonly used utility. For example, to add a 'Mail' menu item to every menu frame, include code similar to the following in ntmenu.tf, just before the 'End' menu item:

  ```
  'Mail' (Validate = 0, Activate = 0, Explanation =
    'Access system mail') =
  BEGIN
    /*4GL code to call the system mail utility */
  END
  ```

  This menu item appears in the top menu only if you can use the substitution variable $default_start_frame:

  ```
  ## IF $default_start_frame THEN
  'Mail' (Validate = 0, Activate = 0, Explanation =
    'Access system mail') =
  BEGIN
    /*4GL code to call the system mail utility */
  END
  ## ENDIF
  ```

- Add user prompts at the end of transactions. For example, you can add a prompt following all Deletes in the Master table field frames which prompts for input before clearing the window and returning from the submenu to the main menu. This reminds the user what the last transaction was before the window is cleared. In this example, code was added to inmtdlmn.tf (Master in table field Delete menuitem), just before the ENDLOOP which exits the UNLOADTABLE loop:

  ```
  IIchar1 = prompt 'You have deleted ' +
  varchar(:IIrowcount)
  + ' rows.  Hit RETURN when you are ready to'
  + ' continue:';
  ENDLOOP;   /*exit submenu */
  ```

- Use an environment variable to control which code is generated:

  ```
  ## IFDEF $$VERBOSE_MODE
  /* 4GL code to run in verbose mode */
  ## ENDIF
  ```

  The generated code can be altered by setting or unsetting the environment variable VERBOSE_MODE at the operating system level.

- Another way of controlling which code is generated is to use substitution variables, for example, your template files can contain code which is dependent on a substitution variable which you have defined:

  ```
  ## IF ('$_mode' = 'verbose') THEN
      /* 4gl code to run in verbose mode */
  ## ELSE IF ('$_mode' = 'expert') THEN
     /* 4gl code to run in expert mode */
  ## ELSE
     /* 4gl code to run in standard mode */
  ## ENDIF
  ```

  Then the code can be generated in three different ways, depending on how you define this environment variable (intopdef.tf is a good place to do this):

  ```
  ## DEFINE $_mode 'verbose' could be 'expert' or some other value
  ```

Including Source Code Processing Commands

Process the source code files Vision generates by specifying commands for Vision to execute after the code is generated. For example, you can check the generated code into a source code library or run your own post-processor over the generated code.

To include source code processing commands, define the logical or environment variable II_POST_4GLGEN to the name of any executable command. After Vision has generated the code for a frame successfully, it checks to see whether II_POST_4GLGEN is defined.

If the logical is defined, Vision then executes the command to which you have defined II_POST_4GLGEN. Vision passes to this command a single argument—the full path name of the 4GL source file that Vision has generated.

Important! Your post-processing code can require alterations to work with subsequent Vision releases.

Editing the Source Code to Create a Custom Frame

You can edit the 4GL code that Vision generates for Menu, Append, Browse and Update frames. However, if you edit the source code directly, Vision marks the frame as a Custom frame.

For a Custom frame, Vision does not regenerate the code when you modify the visual query for the frame and select Go. You must select the Compile operation in the Application Flow Diagram Editor to force code regeneration. When you select Compile for a Custom frame, Vision warns you that compiling the frame overwrites any changes that you have made to the source code. You can continue with the compile and overwrite your source code changes, or you can keep your changes. To protect your edits, save the source code to a new file before selecting Compile, and reapply your edits after regenerating the code.

Vision marks a frame as "Custom" in the application flow diagram whenever the date of the frame's source code file is more recent than the last time the frame's source code was generated. Be aware that the file's date can change if:

- You edit the file and save your changes.
- You save the file before you leave the editor, even if you have not made any changes.
- **VMS:** You exit (rather than quit) from the file, even if you have not made any changes.
- Certain operating system-level changes occur; for example:
  - **UNIX:** You run the touch command on the file.
  - **VMS:** You assign a new owner to the file.

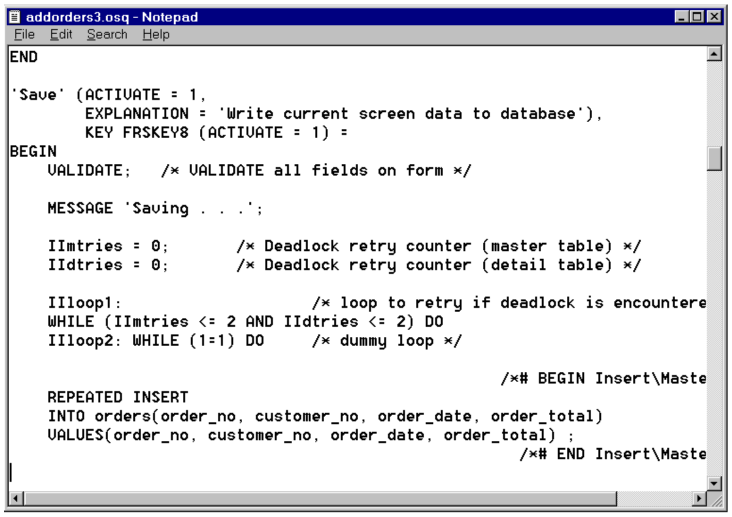
Edit a Source Code File

To edit a Vision source code file, you must use the Edit operation within Vision. Vision marks the frame as Custom as soon as you redisplay the application flow diagram.

If you edit a source code file outside of Vision, the Custom frame status indicator does not appear until the next time Vision tries to regenerate the code for the frame (for example, if you change the frame's visual query).

Vision generates the code for a frame in standard 4GL syntax and format. The source code file includes extensive comments so that you can identify the various components and operations of the frame, including any escape code.

The following figure shows a section of the 4GL source code that Vision generates for an Append frame:



```
END

'Save' (ACTIVATE = 1,
        EXPLANATION = 'Write current screen data to database'),
        KEY FRSKEY8 (ACTIVATE = 1) =
BEGIN
    VALIDATE;   /* VALIDATE all fields on form */

    MESSAGE 'Saving . . .';

    IImtries = 0;       /* Deadlock retry counter (master table) */
    IIdtries = 0;       /* Deadlock retry counter (detail table) */

    IIloop1:                    /* loop to retry if deadlock is encountere
    WHILE (IImtries <= 2 AND IIdtries <= 2) DO
    IIloop2: WHILE (1=1) DO     /* dummy loop */

                                                    /*# BEGIN Insert\Maste
    REPEATED INSERT
    INTO orders(order_no, customer_no, order_date, order_total)
    VALUES(order_no, customer_no, order_date, order_total) ;
                                                      /*# END Insert\Maste
```

To edit the source code file for a Vision-generated frame

1. Open the source code file as follows:
   a. Select Edit from the Application Flow Diagram Editor or Visual Query Editor window for the current frame.
   b. Select Source Code File from the list of edit options.

      Vision opens the source code file for the frame.
2. Use the system editor to modify the code as desired.
3. When you are done, save your changes and exit the file. Vision returns you to the window in which you were working.

Guidelines for Editing Comments in Source Code Files

When you edit a Vision source code file, be careful not to change or delete any of the comment lines that begin as follows:

```
/*# BEGIN
/*# END
```

Also, do not use the comment characters (/*#) to begin any 4GL comments that you add to a source code file.

Vision error processing uses the comment lines beginning with /*# to locate which sections of code contain errors. Editing or deleting any of these lines makes it more difficult to find and edit the errors.

Editing Help Files

Vision creates help files in the application source directory for each frame that it generates. Vision copies the template help file for the frame type and renames it to the name of the frame. The Help files are displayed when a user selects the Help menu operation. Vision generates the Help menu item for all frame types.

The help file that Vision creates in the source directory for a frame is based on the frame's:

- Name

  Each generated help file has the same name as its ".osq" source code file.

- Type

  Vision copies the help file for the frame's type.

- Visual query

  Vision generates a basic help window for each frame type, plus up to two additional help windows based on the frame definition.

Vision can generate the types of help files listed in the following table:

File Extension	Description
.hlp	Contains the primary help text for a frame; appears on the window the user sees when selecting Help from the frame's main menu.
.hlq	Contains the help text for the query-mode menu on a Browse or Update frame; appears in the window that the user sees when selecting Help prior to selecting Go.
.hla	Contains the help text for Update frames on which new records can be appended; appears in the window the user sees by selecting Help after selecting AppendMode.

For example, Vision creates the following help files for an Update frame named ChangeOrders whose source code is in the file "changeorders.osq":

- "changeorders.hlp" If the user selects Help after the user selects Go and is browsing through the data, this file is displayed.
- "changeorders.hlq" If the user selects Help when the user can enter a query before selecting Go, this file is displayed.
- "changeorders.hla" If the user selects Help after the user selects AppendMode, this file is displayed.

Editing Vision-generated Help Files

You can edit any of the generated help files that Vision places in the source code directory. Vision automatically protects your edits so that they are not overwritten the next time you compile the frame.

Provide the Edit Operation

You can edit the help files as you build your application by enabling the Edit menu operation. The Edit operation appears as a menu item in the help window. Selecting this operation in a help window calls the system editor and opens the generated help file for the frame.

To generate the Edit menu operation for help windows in the running application, you must set the logical or environment variable II_HELP_EDIT at the operating system level before running your application. You do this with the following command:

Windows:

```
set II_HELP_EDIT="y"
```

UNIX:

For the C shell:

```
% setenv II_HELP_EDIT "y"
```

For the Bourne shell:

```
$ II_HELP_EDIT="y"
export II_HELP_EDIT
```

VMS:

```
define II_HELP_EDIT "y"
```

Chapter 10: Using Vision Utilities

This section contains the following topics:

This chapter describes the utility functions that Vision provides to help you develop applications. These functions include:

- Calling the operating system or an Ingres tool directly from Vision
- Examining the definitions of tables that you use in your visual queries
- Running a report about an application
- Reconciling your visual queries with any changes to your table definitions
- Cleaning up deleted applications and frames from your disk
- Using alternate sets of global constants

As described in this chapter, most of these functions are accessed by selecting the Utilities operation available on several Vision windows.

Vision also provides a LockStatus utility that reveals any locks that other developers are holding on your applications. This utility is described in detail in the Managing Locks in Vision section of Using Vision or ABF in a Multi-Developer Environment (see page 1381).

Note: Be aware that if your system goes down or you abort a Vision session, Vision regards you as a new user when you start it up again. In this situation, Vision shows you as holding a lock from your original session. You must use the LockStatus utility to clear this lock as described in Managing Locks in Vision or ABF (see page 1386).

Starting Other Programs

While in Vision, you can call the Ingres Menu to start another Ingres tool or return to the operating-system prompt to start an external program without leaving the Vision environment. After exiting the external program, you return to Vision.

Call the Ingres Menu

You can use one of the following options to call the Ingres Menu to access another Ingres tool.

To call Ingres Menu to access another Ingres tool

- Select Utilities from the menu in any of these windows:
  - Applications Catalog window
  - Application Flow Diagram Editor
  - Visual Query Editor
  - Utilities operation in the ABF Frame Catalog window
- Select Ingres/Menu from the submenu.

  The window displays Ingres Menu. You can call any Ingres tool except Vision. You must remain within the same database.
- To return to Vision, select Quit from Ingres Menu.

  Vision displays the window from which you called Ingres Menu.

Return to the Operating System

Use one of the following options to return to the operating system prompt to start another program.

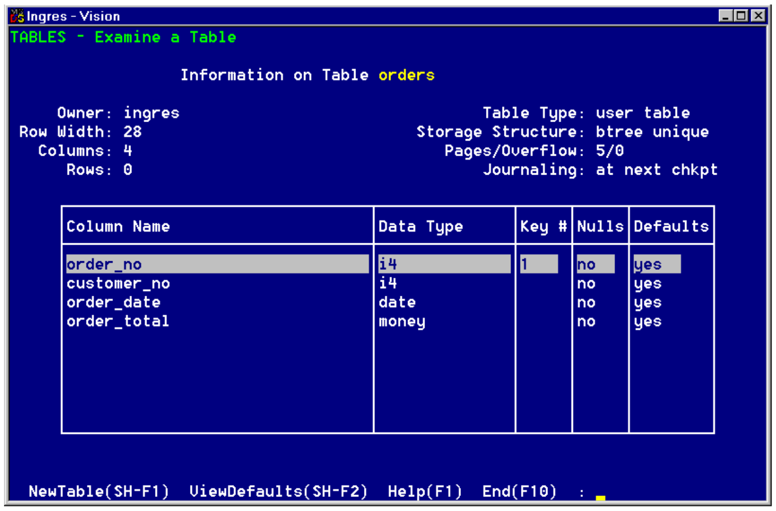
To return to the operating system

1. Select Utilities as described above.
2. Select the appropriate operation for your operating system:
   - **Windows:** Select Shell from the menu.
   - **UNIX:** Select Shell from the menu.
   - **VMS:** Select Spawn from the menu.

   The window displays the operating system prompt.
3. To return to Vision:
   - **Windows:** Type exit at the prompt.
   - **UNIX:** Type exit at the prompt.
   - **VMS:** Type logout at the prompt.

Examining Tables

To get detailed information about the database tables that Vision frames use, display the Examine a Table window, shown in the following figure:



```
Ingres - Vision
TABLES - Examine a Table

                  Information on Table orders

     Owner: ingres                             Table Type: user table
 Row Width: 28                          Storage Structure: btree unique
   Columns: 4                              Pages/Overflow: 5/0
      Rows: 0                                  Journaling: at next chkpt

     | Column Name                      | Data Type      | Key # | Nulls | Defaults |
     | order_no                         | i4             | 1     | no    | yes      |
     | customer_no                      | i4             |       | no    | yes      |
     | order_date                       | date           |       | no    | yes      |
     | order_total                      | money          |       | no    | yes      |

  NewTable(SH-F1)  ViewDefaults(SH-F2)  Help(F1)  End(F10)  :
```

You can display the Examine a Table window:

- In the Application Flow Diagram Editor, to see the definition of a table you want to use in the frame definition
- In the Visual Query Editor, to see the definition of a table you are using in a visual query
- In the Visual Query Editor, to see the definition of a table you want to insert as a Lookup table

The Examine a Table Window

The Examine a Table window displays these fields with information about a table:

- Owner (the account that created the table)
- Width of rows
- Number of columns
- Number of rows
- Table type
- Storage structure
- Number of main and overflow pages
- Whether journaling is enabled or disabled

The window also displays this information about each column in the table:

- Column name
- Data type
- Key #

  If the column is part of a key for the table, the window displays where in the key this column is located. For example, the second column of a multi-column key has #2 displayed in the Key # column.
- Nulls

  The window tells you whether the column can accept null values.
- Defaults

  The window tells you whether the column accepts the Ingres default values (zero for numeric fields, blanks for character fields).

The Vision Examine a Table Window is identical to the Examine a Table Window of the Tables utility. See the *Character-based Querying and Reporting Tools User Guide* for detailed descriptions of the fields displayed in that Examine a Table window.

Display the Examine a Table Window

Use the Examine a Table window to examine a table from the Application Flow Diagram Editor a number of ways.

To display the Examine a Table window while specifying tables for a frame

1. Position the cursor in the Master or Detail field of the table specification pop-up window.
2. Select ListChoices.

   Vision displays the list of available tables.
3. Position the cursor on the name of the table to examine.
4. Select Details.

   Vision displays the Examine a Table window for the table.
5. Select End to redisplay the list of tables.

To examine a table from the Visual Query Editor

1. Position the cursor anywhere on the visual query display of the table you want to examine.

   You can examine the Master or Detail table or a Lookup table.

2. Select TableDef.

   Vision displays the Examine a Table window for the table.

3. Select End to return to the visual query display window.

To examine a table from the Visual Query Editor while inserting Lookup tables

1. Select AddTable.

   Vision displays the pop-up window for specifying the table name.

2. Select ListChoices.

   Vision displays the list of available tables.

3. Position the cursor on the name of the table to examine.

4. Select Details.

   Vision displays the Examine a Table window for the table.

5. Select End to redisplay the list of tables.

While in the Examine a Table window, you can use the NewTable operation to examine additional tables.

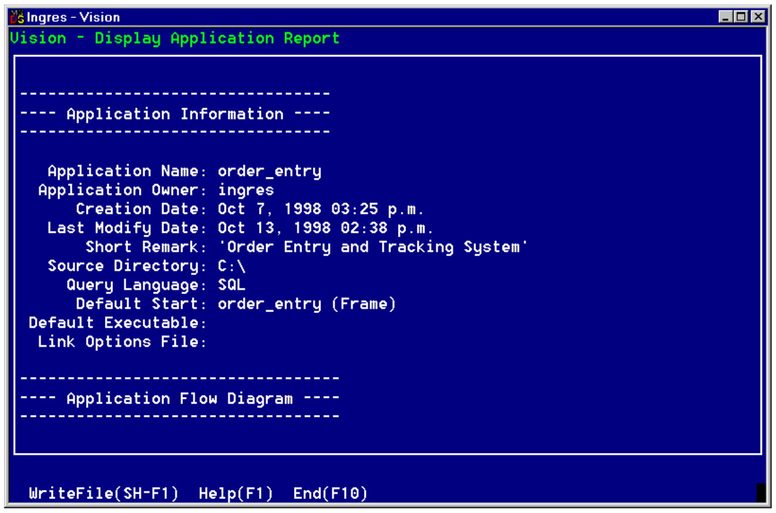
Running an Application Report

Vision provides a report that gives you detailed information about an application. The application report is divided into three sections:

- General information about the application
- The application flow diagram
- Details about each frame in the application flow diagram

General Information Section

The following figure shows the first section of a sample application report:



```
Vision - Display Application Report

--------------------------------
---- Application Information ----
--------------------------------

   Application Name: order_entry
  Application Owner: ingres
      Creation Date: Oct 7, 1998 03:25 p.m.
   Last Modify Date: Oct 13, 1998 02:38 p.m.
       Short Remark: 'Order Entry and Tracking System'
   Source Directory: C:\
     Query Language: SQL
      Default Start: order_entry (Frame)
 Default Executable:
  Link Options File:

---------------------------------
---- Application Flow Diagram ----
---------------------------------

 WriteFile(SH-F1)  Help(F1)  End(F10)
```

The following arfe the fields that the Application Information section contains:

Application Name

Indicates the name of the application

Application Owner

Indicates the name of the account in which the application was created

Creation Date

Indicates the date on which the application was created

Last Modify Date

Indicates the last date in which the MoreInfo About an Application window was edited

Short Remark

Indicates the description that you entered when you created the application

Source Directory

Indicates the directory that contains the application's 4GL source code

Query Language

Always indicates SQL for Vision applications

Default Start

Indicates the frame or procedure that is called when the user starts the application.

This is either the top frame or another frame or procedure that you specify in the Application Defaults window. See Specify a Default Start Frame (see page 281).

Default Executable

Indicates a default image file name that you specify in the Application Defaults window. See Specify a Default Start Frame (see page 281).

Link Options File

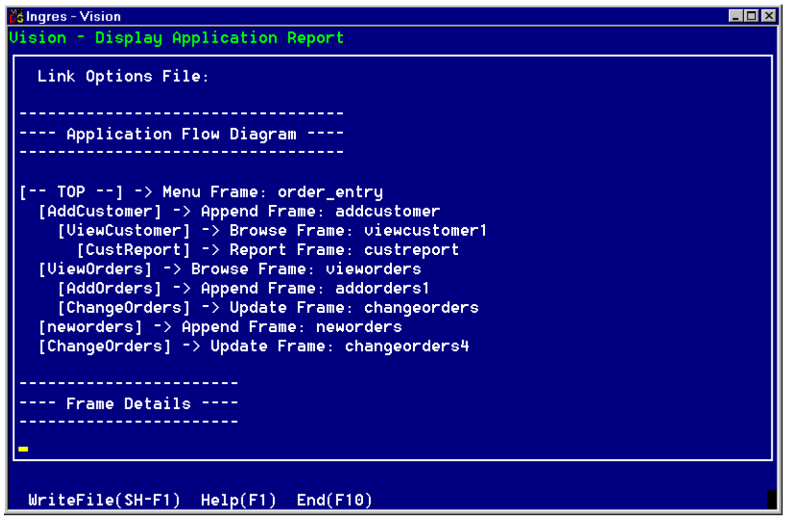
Indicates the file used to link library procedures that are not maintained through Vision. See the Specify the Link Options File section in Define a 3GL Procedure (see page 171).

Application Flow Section

The second part of the report lists the frames in the application flow diagram. The list is indented to represent the tree structure of the application as shown in the following figure.

For each frame, the list includes:

- The menu item to call the frame
- The frame type
- The frame name



```
Vision - Display Application Report

  Link Options File:

----------------------------------
---- Application Flow Diagram ----
----------------------------------

[-- TOP --] -> Menu Frame: order_entry
  [AddCustomer] -> Append Frame: addcustomer
    [ViewCustomer] -> Browse Frame: viewcustomer1
      [CustReport] -> Report Frame: custreport
  [ViewOrders] -> Browse Frame: vieworders
    [AddOrders] -> Append Frame: addorders1
    [ChangeOrders] -> Update Frame: changeorders
  [neworders] -> Append Frame: neworders
  [ChangeOrders] -> Update Frame: changeorders4

-----------------------
---- Frame Details ----
-----------------------

 WriteFile(SH-F1)  Help(F1)  End(F10)
```

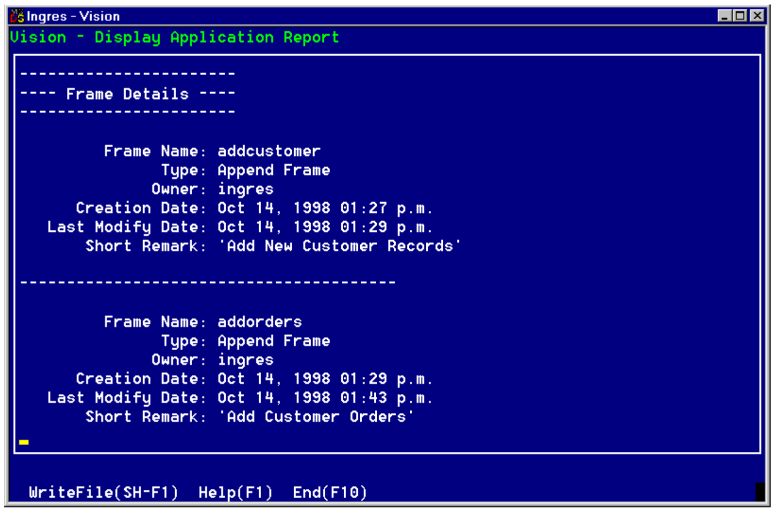
Frame Detail Section

The third part of the report contains detailed information about each frame in the application, including any you have removed from the application flow diagram (but not those you have destroyed).

For each frame listed, the report displays:

- Frame Name
- Type
- Owner (the user who created the frame)
- Creation Date
- Last Modify Date
- Short Remark describing the frame

The following figure shows an example of the Frame Details section of the Application Report:



```
Ingres - Vision
Vision - Display Application Report

-----------------------
---- Frame Details ----
-----------------------

         Frame Name: addcustomer
               Type: Append Frame
              Owner: ingres
      Creation Date: Oct 14, 1998 01:27 p.m.
   Last Modify Date: Oct 14, 1998 01:29 p.m.
       Short Remark: 'Add New Customer Records'

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

         Frame Name: addorders
               Type: Append Frame
              Owner: ingres
      Creation Date: Oct 14, 1998 01:29 p.m.
   Last Modify Date: Oct 14, 1998 01:43 p.m.
       Short Remark: 'Add Customer Orders'

  WriteFile(SH-F1)  Help(F1)  End(F10)
```

View an Application Report

To view an application report

1. Position the cursor on the application name in the Applications Catalog window.
2. Select Utilities from the menu.

   Alternatively, you can select Utilities from the Application Flow Diagram Editor.
3. Select AppReport from the menu.

Vision runs the report and displays it to the window. To save the report to a file, see Write an Application Report to a File (see page 251).

4. Select End to return to the window from which you called the report.

Write an Application Report to a File

You cannot edit the application report when Vision displays it in the window. However, you can write the report to a file that you can edit and print.

To write an application report to a file

1. Display the report in the window as described in the previous section. When Vision displays the report, select WriteFile from the menu.
2. When prompted, enter the name of a file to which to write the report.

   If the file is not in the current directory, enter the full directory path.
3. Press Return.

   Vision writes the report to the file you specify.
4. You can do either of the following:
   - Continue to view the report.
   - Select End to return to the window from which you called the report.

Reconciling Tables and Frame Definitions

After you begin to develop a Vision application, you can change a table used in your visual queries; (for example, adding a column or specifying a different data type for a column) by recreating the table in the Tables Utility with a different definition (see *Character-based Querying and Reporting Tools User Guide* for details on using the Tables utility).

If you change a table, the visual queries for any frames that use this table do not reflect the changes automatically. However, Vision provides a utility to reconcile frame definitions to match any changes made to the tables they use.

You can use the Reconcile utility with any frames in an application, including those that you have removed from the application flow diagram. The Reconcile utility:

- Checks for discrepancies between the tables in the database and the definitions of the frames that you specify
- If it finds any discrepancies, lets you tell Vision to change the frame definition to match the table structure or to reject the changes

The Reconcile utility operates on a "per frame" basis. You must accept or reject all changes to a frame; that is, if there are multiple changes to a frame, you cannot accept some and reject others.

Note: If the changes to your application are fairly complex, you can reject them when you run the Reconcile utility and write the reconciliation report to a file. After studying the changes that Vision proposes and considering their impact on your application, you can accept the changes by running the Reconcile report again.

You can use the Reconcile utility to experiment with your tables. You can change a table and run the utility for all frames that use the table. The Reconcile utility shows you how these changes affect the frame definitions. You can view the report and write it to a file, and reject the changes so that Vision does not make the changes permanent.

Also, if other developers are using the same tables, you can run the Reconcile utility periodically to check on any changes to tables that they have made that affect your applications and frames.

How Vision Reconciles Frame Definitions

When you change a table and run the Reconcile utility, Vision makes the following changes to the visual queries for all frames that use that table:

If You Make This Change to a Table:	Then Vision Makes This Change to the Frame Definition:
Delete a Master or Detail table from the database	Vision cannot modify the frame definition.
Delete a Lookup table from the database	Deletes the Lookup table from visual query.
Delete a column from a table	Deletes the column from the visual queries. Vision also deletes any escape code using this column and displays the escape code in the reconciliation report. If the column you delete is used as a join column, Vision deletes the join.
Add a column to a table	Adds the column as non-displayed to the visual queries.
Change the data type or nullability of a column	Changes the column's data type or nullability in the visual queries. If you change the data type of a join column so that the join is no longer valid, Vision deletes

If You Make This Change to a Table:	Then Vision Makes This Change to the Frame Definition:
	the join and notes this fact in the reconciliation report.
Change the key columns for a table	Adjusts the visual queries as necessary.

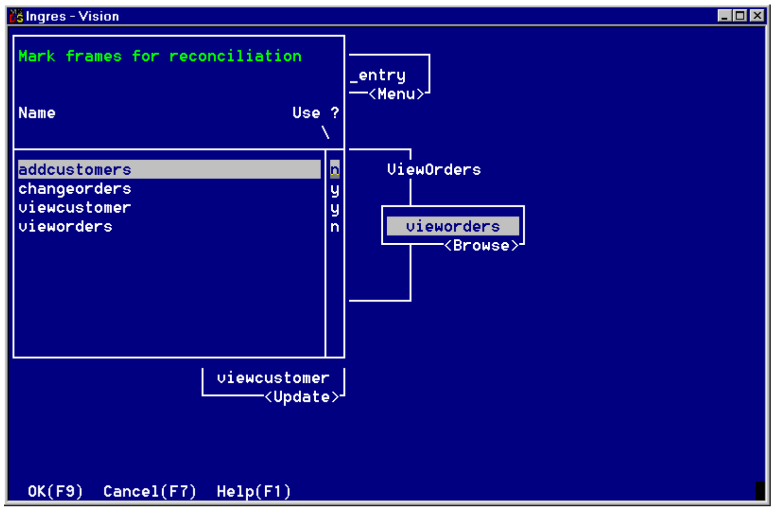
The reconciliation report notes whether any changes cause a visual query to be invalid; for example, by deleting a join column. Also, you must re-specify column titles for Lookup table selection lists after you run the Reconcile utility.

Run the Reconcile Utility

To reconcile tables and frame definitions for an application

1. Select Utilities in either of these ways:
   - With the cursor in the name of the application on the Applications Catalog window
   - From the Application Flow Diagram Editor for the application
2. Select Reconcile.

   Vision displays a pop-up window, shown in the following figure, that lists all the frames in your application. You use this list to specify the frames to be reconciled.



```
Mark frames for reconciliation

Name                            Use ?

addcustomers                     n
changeorders                     y
viewcustomer                     y
vieworders                       n
```

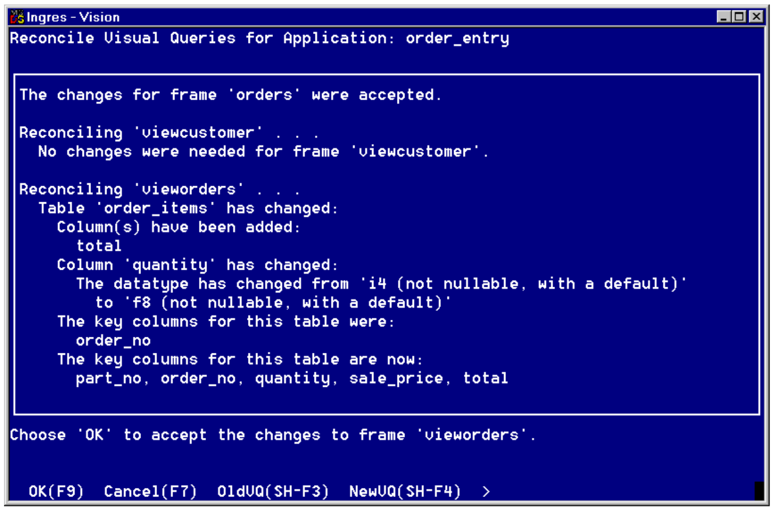
   By default, Vision marks all frames as "y," indicating that they must be reconciled.

3. Specify any frames that you do not want to reconcile by typing n in the Use? field next to the name of each frame.

4. After you specify the frames you want to reconcile, select OK.

   Vision displays the reconciliation report for the frames you specify. The following figure shows a portion of a sample reconciliation report window.

   For each frame that can be changed, Vision asks you to accept or reject the changes. Before responding, you can use the OldVQ operation to view the old visual query or the NewVQ operation to see the proposed new visual query.



```
Reconcile Visual Queries for Application: order_entry

 The changes for frame 'orders' were accepted.

 Reconciling 'viewcustomer' . . .
   No changes were needed for frame 'viewcustomer'.

 Reconciling 'vieworders' . . .
   Table 'order_items' has changed:
     Column(s) have been added:
       total
     Column 'quantity' has changed:
       The datatype has changed from 'i4 (not nullable, with a default)'
         to 'f8 (not nullable, with a default)'
     The key columns for this table were:
       order_no
     The key columns for this table are now:
       part_no, order_no, quantity, sale_price, total

Choose 'OK' to accept the changes to frame 'vieworders'.
```

5. Accept or reject the changes as follows:

   - Select OK to accept the changes.

     Vision makes the changes. You must accept all changes for the frame.

   - Select Cancel to reject the changes.

   Vision continues to run the report.

6. After the report is finished, you can write it to a file as follows:

   a. Select WriteFile from the menu.

      Vision prompts you for the name of a file.

   b. Enter the file name and press Return.

      If the file you specify is not in the current directory, be sure to include the full directory path.

7. Select End to return to the window from which you called the Reconcile utility.

If you have not written the report to a file, do so before you exit the Reconcile utility.

Clean Up a Database

As you develop your Vision applications, you can destroy database objects—frames, forms, or entire applications—that you no longer need. When you destroy these objects, you no longer can use them.

However, these objects still exist in the Ingres system catalogs in the database and take up space on your disk. Vision provides a Cleanup utility that checks the system catalogs for database objects that you have destroyed. It removes the catalog entries for these objects to create space for new objects.

You must be the database administrator for the database to use the Cleanup utility. It is good practice to run the Cleanup utility periodically and after a number of database objects have been destroyed.

However, be aware that the utility cleans up the entire database and, therefore, can tie up your system for a considerable amount of time. Run the utility when no one needs access to the database.

To run the Cleanup utility

1. Select Utilities from the menu on the Applications Catalog window or Application Flow Diagram Editor.

   Because the utility affects the entire database, you can run it from any application.
2. Select Cleanup from the submenu.

   (This menu item appears only if you are the database administrator for the database in which you are working.)

   Vision displays a warning that the utility takes a while to run and asks you to confirm that you want to run it.
3. Confirm that you want to run the Cleanup utility.

   Vision runs the utility; when it finishes, it displays the number of rows deleted.
4. Select End to return to the window from which you called the Cleanup utility.

You also can clean up a database with the sysmod utility that releases the recovered disk space back to the operating system.

Note: Visual DBA provides an alternative way to clean up the database. Detailed steps for performing this procedure can be found in the Dropping Objects section of the online help for Visual DBA.

Use Alternate Sets of Global Constants

Your application normally uses the default set of constants defined in the application. You can set up alternate sets of global constants. For example, you can define and use constants for all language-dependent strings you use in an application. You can create one set for each language you want to use.

The default set of constants is stored in the database. You can create alternate sets of constants and store them in files. You can override the default constants by specifying the file name before running or editing an application. You can also specify a constants file when you create an image of the application.

When a global constant file is specified before running the application or creating an image, the values in the constant file override the default values stored in the database for that operation only. The contents of the file do not affect any subsequent Image, Go or Edit operations.

To specify a global constants file

1. Select Utilities from the menu in the Applications Catalog window or Application Flow Diagram Editor.
2. Select ConstantsFile from the menu.
3. Specify a filename in the Global Constants File Name field.
4. Return to the Applications Catalog or Application Flow Diagram Editor.
5. Select Go or Edit to run the application with the specified constants file.

Create an Alternate Global Constants File

Create a global constants file with your system editor, or save the default global constants (that you defined through the Edit Global Constants window) into a file.

To create an alternate set of global constants from the default set

1. Select Utilities from the menu in the Applications Catalog window or Application Flow Diagram Editor.
2. Select ConstantsFile from the menu.
3. Specify a filename in the Global Constants File Name field.
4. Select Create to save the default global constants into a file.

To create an alternate set of global constants with your system editor, create and edit a file with the following format:

- A constant name, which must begin in column 1.
- The character : or =.

  White space can appear on either side of this character.
- A value for the constant.

  The value can continue across multiple lines, but continuation lines must begin with a space, or tab.

For example:

```
menu1: 'This is the first menu item'
menu2 = 'Menu2 is the second menu item'
```

The file can also contain comments beginning with a # character.

Edit an Alternate Set of Global Constants

Change the alternate set of global constants file by editing the file. You cannot edit the alternate set of constants through the Edit Global Constants window. Edit the file with your system editor, from your operating system or from within Vision.

To edit the file from within Vision

1. Select Utilities from the menu in the Applications Catalog window or Application Flow Diagram Editor.
2. Select ConstantsFile from the menu.
3. Specify a filename in the Global Constants File Name field.
4. Select Edit from the menu to edit the specified file.

When you edit or destroy a constant in one file, it does not affect the constants in another file, or the default values in the database.

Chapter 11: Completing a Vision Application

This section contains the following topics:

This chapter describes the steps you take after you have created the frames and specified the queries for your application. These steps include:

- Testing applications and fixing any errors
- Regenerating source code and forms
- Editing errors in generated code
- Letting users access the application conveniently from the operating system prompt, without having to call Vision; you do this by:
  - Creating an executable image of the application
  - Creating a command file for users to run the application
- Running completed applications
- Renaming applications
- Copying applications to another database
- Copying application components
- Destroying applications you are no longer using

Test Applications

After testing your application at various stages of development, you are ready to test the final product.

When you test an application, Vision regenerates the code and forms for the parts of the application that have been modified since the last time you ran it.

When testing your application, you can test:

- An entire application from the Applications Catalog window
- All or part of an application from the Application Flow Diagram Editor

To test an entire application from the Applications Catalog

1. Position the cursor on the name of the application.
2. Select Go.

   Vision updates the code and forms for the frames of the application that you have changed, then runs the application.

To test an application from the Application Flow Diagram Editor

1. Select the current frame where you want to begin testing the application. To run the whole application, make the top frame the current frame.
2. Select Go.

   Vision tells you for which parts of the application it is regenerating forms or 4GL code. Vision updates the code and forms as necessary for the frames you select, then runs the application for the current frame and any frames beneath it.

After you are done testing the application, you are returned to the window from which you selected Go.

To regenerate the code and forms for a specific frame but not run the frame, select Compile from the Application Flow Diagram Editor menu instead of Go. Use the Compile operation to verify that you have correctly specified or modified a frame.

The next section discusses the circumstances that cause Vision to regenerate the code and form for a frame.

Regeneration of Source Code and Forms

Vision regenerates the 4GL code and form for a frame because you have:

- Newly created the frame
- Changed the visual query; for example, specified that different columns be displayed as form fields, modified frame behaviors or added escape code

- Inserted or removed child frames from the application flow diagram (Vision regenerates the code and form for the parent frame to add or delete the appropriate menuitems)

  If you select Go, Vision does not regenerate the form for a Menu frame if you have used the forms editor to modify the form. You must select the Compile operation to regenerate the form for such a frame.
- Changed a global component or local variable that the frame uses
- Changed the return type of a child frame or called procedure

How Custom Frames Are Handled

Vision marks a frame as "Custom" in the application flow diagram whenever the date of the frame's source code file is more recent than the last time the frame's source code was generated. Be aware that the file's date can change if:

- You edit the file and save your changes.
- You save the file before you leave the editor, even if you have not made any changes.
- **VMS:** You exit (rather than quit) from the file, even if you have not made any changes.
- Certain operating system-level changes occur; for example:
  - **UNIX:** You run the touch command on the file.
  - **VMS:** You assign a new owner to the file.

Vision does not regenerate the code for a Custom frame when you select Go. You must select the Compile operation in the Application Flow Diagram Editor to force code regeneration of a Custom frame. Vision warns you that compiling the frame can cause any changes that you have made to the source code to be overwritten.

Note: To edit a Vision source code file, use the Edit operation within Vision. Vision marks the frame as Custom when you redisplay the application flow diagram. For more information, see Modifying Vision Code (see page 227).

When editing a source code file outside of Vision, the Custom frame status indicator does not appear until the next time Vision tries to regenerate the code for the frame (for example, if you change the frame's visual query).

Summary of Code Regeneration Operations

The following table summarizes the Vision operations that affect the regeneration of source code and forms:

Operation	Effect
Go Image Edit Data Entry Form (except for Menu frames)	Causes regeneration only if needed (as described above)
Compile	Always forces regeneration See the Handling Custom Frames section for information about compiling Custom frames.
Edit Source Code File (so that the frame becomes a Custom frame as described above)	The frame is marked as Custom in the application flow diagram; you must select Compile to force Vision to regenerate the code
Edit Data Entry Form (Menu frames only)	You must select Compile to force Vision to regenerate the form

Editing Errors in Generated Code

The 4GL code that Vision generates for an application generally does not contain any syntax or other errors. However, when you compile or test an application, Vision detects errors in the code if you:

- Specify an invalid expression in a default value or assigned value on an Append frame or a query restriction on an Update or Browse frame
- Write escape code that contains errors

  Vision does not check your escape code for errors until it compiles it with the code that it generates for the frame.
- Specify any parameters or local variables incorrectly

If Vision finds any errors in the code it generates for a frame, it marks the frame with the "Error" indicator in the application flow diagram. This indicator remains until you correct the code and recompile the frame.

In addition, Vision generates a listing file that contains all errors for each frame you have compiled. Errors remain in this listing file until you correct them and recompile the code.

Use the Vision error handler described below to identify and correct any errors in your Vision generated source code. You can call the error handler at any time after the error occurs.

Edit the source code files directly. However, it is more convenient to use the error handler facility to track and fix your errors, because the error handler shows you the specific location of the error.

Call the Error Handler

Call the error handler in either of these ways:

- While compiling a frame that contains incorrect code
- While using the Application Flow Diagram Editor or Visual Query Editor

To call the error handler while compiling a frame that contains incorrect code

1. On the application flow diagram display, select Compile for the current frame.

   If the code contains any errors, Vision generates an error message and asks if you want to edit the errors.

2. Enter yes and press Return.

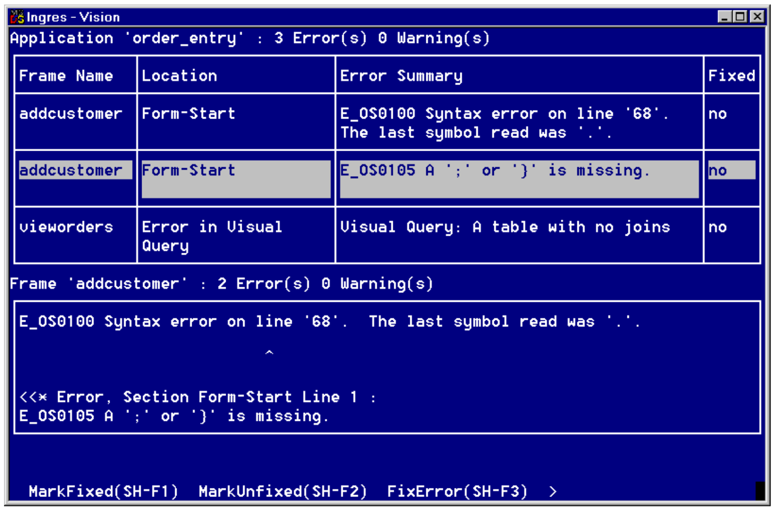
   Vision displays the error window.

To call the error handler from the Application Flow Diagram Editor or Visual Query Editor

1. With any frame as the current frame, select Edit from the menu on the application flow diagram or visual query display.
2. Select Errors from list of edit options.

   Vision displays the error window.

The following figure shows an example of the error window:



```
Application 'order_entry' : 3 Error(s) 0 Warning(s)
```

Frame Name	Location	Error Summary	Fixed
addcustomer	Form-Start	E_OS0100 Syntax error on line '68'. The last symbol read was '.'.	no
addcustomer	Form-Start	E_OS0105 A ';' or '}' is missing.	no
vieworders	Error in Visual Query	Visual Query: A table with no joins	no

```
Frame 'addcustomer' : 2 Error(s) 0 Warning(s)

E_OS0100 Syntax error on line '68'.  The last symbol read was '.'.
                         ^

<<* Error, Section Form-Start Line 1 :
E_OS0105 A ';' or '}' is missing.


  MarkFixed(SH-F1)  MarkUnfixed(SH-F2)  FixError(SH-F3)  >
```

Moving Around in the Error Handler

When you call the error handler, the error window helps you identify your errors and provides operations for correcting them. The error window shows all uncorrected errors for an entire application.

The top part of the error window contains a table field with the following columns:

Frame Name

Lists in alphabetical order all the frames in the application with errors

If you call the error handler while compiling a frame, that frame's errors appear at the top of the list.

Location

Provides a general description of where the error occurred

In the preceding figure, for example, "Form-Start" indicates that you incorrectly specified Form-Start escape code.

You can use the standard Top, Bottom or Find operations to scroll through the errors listed or to locate a specific frame name.

Error Summary

Provides a brief description of the error and, where applicable, provides the standard Ingres error number and message

Fixed

Used to note corrected errors until you recompile a frame

The center part of the error window contains:

- The name of the frame on which the cursor is positioned
- The number of errors for this frame (this includes errors that are marked as Fixed but which have not been recompiled yet)

The lower part of the display contains a window that displays up to five lines of the listing file (described in View the Error Listing File (see page 267)) for the error on which the cursor is positioned. As you scroll through the errors listed at the top of the window, Vision displays the corresponding section of the error listing file in the window.

Error Handler Menu Operations

The error window provides the following menu operations:

MarkFixed

Marks an error as fixed in the table field at the top of the window

MarkUnfixed

Marks an error as not fixed in the table field at the top of the window

FixError

Displays the specific location where the error occurred

Compile

Recompiles a frame

ListingFile

Displays the full listing file containing all current errors for a frame

Help

Provides on-line help about how to use the error handler

End

Returns you to the window in which you were working or resumes compiling a frame (depending on how you called the error handler)

The following sections describe how to use these operations to track and correct your errors.

Fix Errors in the Generated Code

The error handler lets you view and correct errors in a frame's 4GL code. Errors generally are located in one of these areas:

- The Visual Query Editor window
- The parameter or local variable definition window
- Escape code you have entered for a frame

When you use the FixError operation as described below, Vision displays as precisely as possible the location where the error occurred. For example, if you have specified an incorrect query restriction on an Update frame, Vision displays the section of the visual query window containing the incorrect specification.

You must return to the original location to correct certain errors; you cannot go through the error handler. For example, if you have specified a frame definition without a join column between the Master and Detail table, you must return to the Visual Query Editor to add a join column.

You can identify such errors because their error listing file is specified as "Unavailable." Vision generates an error message if you try to fix the error through the error handler.

Remember that a single syntax error in 4GL code—omitting a statement terminator in escape code, for example—can generate a series of subsequent syntax errors in the code. You can identify when this happens because the errors appear in successive lines of code. Usually, fixing the first error also corrects the errors that follow.

To fix an error through the error handler

1. PositionFix Errors in the Generated Code the cursor on the error at the top of the error window.
2. Select FixError from the menu.

   Vision displays a pop-up containing the incorrect specification or code.
3. Correct the error.
4. Select End to save your changes and return to the error window.

The error now contains a value of "yes" in the Fixed field. Vision also marks the error as "FIXED" in the error listing file for the frame.

These indicators do not necessarily mean that you have fixed the error correctly. Vision cannot determine that the error is corrected until you recompile the frame.

If you know that you have not corrected the error, you can change the designation in the Fixed field to "no" as described below.

Recompile a Frame

After you attempt to correct an error in the source code for a frame, you must have Vision regenerate the code so that you can verify your changes. You can recompile a frame directly from the error window.

When you recompile a frame, the error handler:

- Removes all successfully corrected errors on the frame from the error listing file and the error window
- Generates a new error listing file with any uncorrected errors for the frame
- Lists the uncorrected errors with a value of "no" in the Fixed column, even if you previously had marked them as fixed

To recompile a frame from the error handler

1. Position the cursor on any error for the frame you want to recompile.
2. Select Compile from the menu.

   Vision generates a new form and source code for the frame. The error listing file and error window are adjusted as described in the Fixing Errors in the Generated Code section.

View the Error Listing File

Each time you compile a frame, Vision generates a new error listing file. As an alternative to viewing each error individually in the error window, you can view the entire error listing file for a frame. You must view the listing file:

- To see all errors for a frame

  This ability is particularly helpful in the case of false syntax errors (sequential errors that result from a single incorrect syntax specification, such as a missing statement terminator)
- When the code and its error explanation are too long to fit in the error window

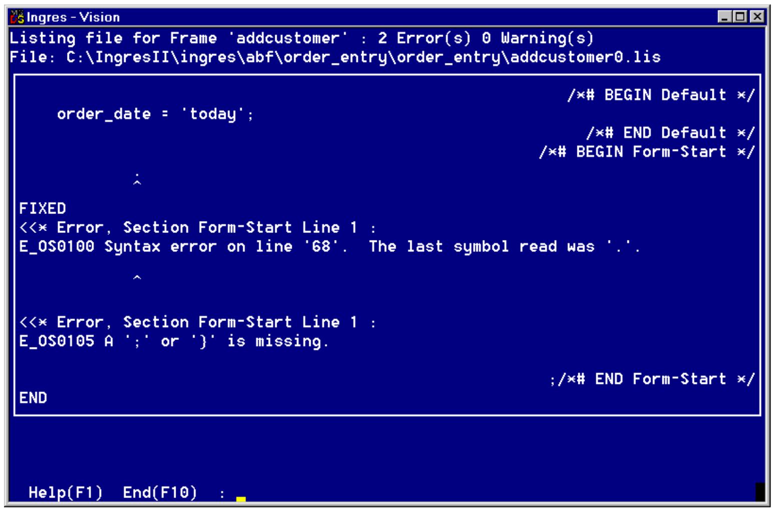
The following figure shows the error listing file for a frame. If you are familiar with writing 4GL code, the file must look similar to the standard 4GL error listings. The Vision error listing file also uses these unique display conventions:

- The symbol "<<" indicates the description of the error
- The comment indicator (/*#) marks the beginning and end of the section of generated code that contains the error

  The comment corresponds to the location of the error as listed in the error window. For example, the following section of a listing file indicates incorrect Form-Start escape code:

  ```
     /*# BEGIN Form-Start */
  Form-Start escape code...
     /*# END Form-Start */
  ```

The error listing file is a read-only file. You cannot correct any errors in this file.



```
Listing file for Frame 'addcustomer' : 2 Error(s) 0 Warning(s)
File: C:\IngresII\ingres\abf\order_entry\order_entry\addcustomer0.lis
                                                         /*# BEGIN Default */
    order_date = 'today';
                                                           /*# END Default */
                                                      /*# BEGIN Form-Start */
            ;
            ^
FIXED
<<* Error, Section Form-Start Line 1 :
E_OS0100 Syntax error on line '68'.  The last symbol read was '.'.
            ^

<<* Error, Section Form-Start Line 1 :
E_OS0105 A ';' or ')' is missing.

                                                       ;/*# END Form-Start */
END

  Help(F1)  End(F10)  :
```

To view the error listing file for a frame

1. Position the cursor on any error for the frame in the error window.
2. Select ListingFile from the menu.

   Vision displays the current error listing file for the frame.

Mark Errors as Fixed or Unfixed

The Fixed field in the error window lets you keep track of the errors that you have fixed. You can use this field as a record until the next time you recompile a frame. The value of this field has no effect on the actual status of the 4GL source code containing the error.

Whenever you use the FixError operation, Vision marks the error as fixed in the error window, even though you cannot actually have fixed the error. If this happens, you can change the value of Fixed back to "no" to remind yourself that you have not recompiled the frame successfully.

You also can fix an error outside of the error handler; in the visual query window, for example. The error handler has no knowledge of your correction, and leaves the error indication as unfixed.

You can manually change the status indicator of an error with the MarkFixed and MarkUnfixed operations.

To change the fixed status of an error

1. Position the cursor on the error at the top of the error window.
2. Select MarkFixed or MarkUnfixed from the menu as appropriate.

   The value in the Fixed field for the error changes to "yes" or "no," respectively.

If you recompile a frame and an error remains that you have marked as fixed, its fixed status reverts to "no."

Making Applications Easier to Run

Users can access the application conveniently from the operating system prompt, without having to call Vision. You do this by:

- Creating an executable image of the application
- Creating a command file for users to run the application

Create an Image of an Application

When your application is ready to use, you can create an executable *image* of the application. An image is a version of the application that the user can run directly, without going through Vision.

Although building the image can take a considerable amount of time, it makes the application easier to use. Users do not need to know how to access Vision. Instead, they can call the application directly by the name of the image file.

Also, the image file serves as a "snapshot" of the application at the time the image is created. Thus, you can save a version of the application while you continue to build it in Vision.

Windows: You must have embedded SQL for C available to image an application.

To create the image of an application

1. In the Applications Catalog window, move the cursor to the name of the application and select Utilities from the menu.

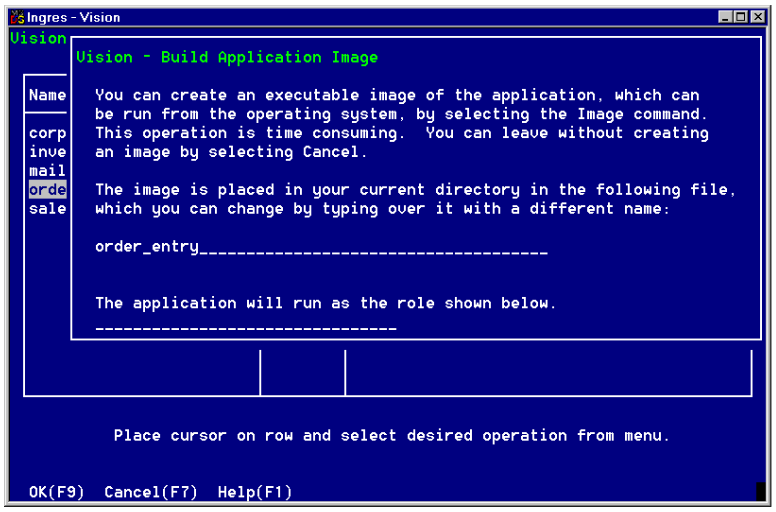
   Alternatively, if you currently are working with the application you want to image, you can select Utilities from the Application Flow Diagram Editor menu.

2. If you want to change the global constants file, do so now. For more information, see the Using Alternate Sets of Global Constants section in Using Vision Utilities (see page 243).

3. Select Image from the submenu.

   Vision displays a pop-up, shown in the following figure, that displays the name of the default image file. This name is the same as the application, unless you specify a different file as described in the Specifying a Default Image File section. For Microsoft Windows and VMS, Vision gives this file the extension .exe.

   By default, Vision places the image file in the directory in which you currently are working. To accept the default file specification, proceed to Step 5.



```
Ingres - Vision
Vision
        Vision - Build Application Image

Name      You can create an executable image of the application, which can
          be run from the operating system, by selecting the Image command.
corp      This operation is time consuming.  You can leave without creating
inve      an image by selecting Cancel.
mail
orde      The image is placed in your current directory in the following file,
sale      which you can change by typing over it with a different name:

          order_entry_____________________________________


          The application will run as the role shown below.
          ________________________________

            Place cursor on row and select desired operation from menu.

  OK(F9)  Cancel(F7)  Help(F1)
```

4. To change the default file name and/or directory, type in a new specification.

   If you are changing the directory, be sure to enter the full directory path.

   Specify a role identifier. The Knowledge Management Extension uses the role identifier to associate specific permissions with the application.

   See the *SQL Reference Guide* for more information about roles.

5. Select OK from the menu.

   Vision generates the necessary code, constructs the image file, and places it in the specified directory.

Specify a Default Image File

Use the Application Defaults window to specify an image file name that is different from the application name. The name you specify appears as the default image file when you select the Image operation as described previously.

To specify a default image file

1. Position the cursor on the name of the application in the Applications Catalog window.
2. Select MoreInfo from the menu.

   Vision displays the MoreInfo About an Application window.
3. Select Defaults from the window.

   Vision displays the Application Defaults window.
4. Move the cursor to the Default Image Name field.
5. Enter the name of the new default image file.

   If you do not intend the file to be in the current directory, be sure to enter the full directory path specification.
6. Select OK to return to the MoreInfo About an Application window.
7. Select Save.
8. Select End to return to the Applications Catalog window.

Specify an Application Role Identifier

Specify a role identifier in the Application Defaults window. The Knowledge Management Extension uses the role identifier to associate specific permissions with the application.

See the *SQL Reference Guide* for more information about roles.

To specify a role identifier

1. Position the cursor on the name of the application in the Applications Catalog window.
2. Select MoreInfo from the menu.

   Vision displays the MoreInfo About an Application window.
3. Select Defaults from the window.

   Vision displays the Application Defaults window.
4. Move the cursor to the Application Role field.

5. Enter the name of a role identifier that exists in the database.
6. Select OK to return to the MoreInfo About an Application window.
7. Select Save.
8. Select End to return to the Applications Catalog window.

imageapp Command—Create a Command File

Build an image from the operating system level by using the imageapp command. This command builds an executable image the same way that the Image menu item does, but uses a command-line syntax instead of the forms-based interface. Include the imageapp command in a command file or a batch file.

This command has the following format:

```
imageapp [v_node::]dbname applicationname [-uusername][-f]
      [-w][-5.0][+wopen][-oimagename][-GgroupID][-Rrolename]
      [-constants_file='filename']
```

[*v_node*::]*dbname*

Specifies the name of the database.

If you are using a database that resides on a remote host, you must specify the *v_node,* followed by two colons. For example*:*

```
server1::orderdb
```

applicationname

Specifies the name of the application from which to build an image.

If not specified, Vision prompts you for the application name.

-u*username*

Runs the application as if you were the user represented by *username*.

Files created under this flag are owned by the user actually running the Vision process, not by *username.*

To use this option, you must be a privileged user.

If you are using Enterprise Access Products (formerly Gateways), refer to your Enterprise Access documentation before using this parameter.

-f

Forces Vision to recompile the 4GL code for the entire application, even if no changes have been made since the last time it was compiled.

-w

Causes all warnings to be turned off. For more information, see Disabling Warnings (see page 1382).

-5.0

Invokes 4GL in a mode that is compatible with Ingres Release 5. For more information, see 5.0 Applications (see page 1383).

+wopen

Generates warnings if Vision detects statements that are not compatible with OpenSQL

-o*imagename*

Specifies the name for the image. If the -o flag is not specified, Vision uses the default image name specified in the Applications Defaults window. If no default image name is specified, the image is given the same name as the application.

-G*groupid*

VMS: Lets you run or edit the application as a member of the group specified.

Capital letter flags require double quotes in VMS, for example: "**-**G*groupid*"

-R*rolename*

VMS: Assigns a role to an application image.

If you specify a role name, you are prompted for the role's password. Roles are a feature of the Knowledge Management Extension.

Capital letter flags require double quotes, for example: "**-**R*rolename*"

-constants_file='*filename*'

Specifies a file containing values for the application's constants. If the -constants_file flag is specified, the values in the constants file override the values for the constants stored in the application.

The *filename* can be the full directory path name for the constants file.

Example: imageapp command

To recompile all frames and create an image called "orders" for the Order_entry application in the Inventory database, enter:

```
imageapp inventory order_entry -oorders -f
```

imagename Command—Run a Completed Application

After you create the image, your application is ready to use. Users can run the image at the operating system by specifying the image name and the desired parameters.

Note: To make things easier for the user, and to allow more control over how the application is used, you can also create a command to include the application image and the parameters. See Creating a Command for an Imaged Application (see page 280) for details.

VMS: To run the image, use run *imagename*. Because the run command only accepts one argument, the executable name, you must define the image as a DCL foreign command to run the image with any of the other parameters described in this section. See Creating a Command for an Imaged Application (see page 280) for details.

This command has the following format:

```
imagename  [ -d[v_node::]dbname | -database=[v_node::]dbname
             | -nodatabase ] [-uusername] [framename |[-p]procname]
             [-noforms |-forms] [SQL option flags] [-Ggroupid] [-Rrolename]
             [-constants_file='filename'][-a application_specific_parameter
             {application_specific_parameter}]
```

imagename

Specifies the name of the application image. By default, the image name is the same as your application name.

-d[*v_node*::]*dbname* |

-database=[*v_node*::]*dbname*

Runs the application with the database specified by *dbname*.

The -database flag and the -d flag are synonymous.

The -database or -d flag lets you run the application with a database other than the one the application resides in. The new database must contain the same tables and reports as those used in the application's queries. The new database must also contain the necessary forms, if the application does not use compiled forms.

For example, you can develop an application on a test database and later run it on a production database.

If you are using a database that resides on a remote host, you must specify nodename, followed by two colons. For example:

```
-dserver1::orderdb
```

-nodatabase

Starts the application without an open database session. The **-**nodatabase flag can be used to run an application that does not require access to a database, or an application that starts a database session with the 4GL connect statement.

See the 4GL reference part of this guide for information about database connections.

-u*username*

Runs the application as if you were the user represented by username

Files created under this flag are owned by the user actually running the ABF process, not by username.

To use this option, you must be a privileged user.

If you are using Enterprise Access Products, refer to your Enterprise Access documentation before using this parameter.

framename

Runs the application with the specified frame as the top frame

[-p]*procname*

Runs the application beginning with the procedure represented by procname.

You only need to include the -p flag before the procedure name if you are invoking an image created under a previous release of Vision and the procedure has the same name as a frame in the application.

-noforms|-forms

The -noforms flag lets you run an imaged application without initializing the Forms Runtime System. If the application attempts an operation that requires the forms system, a runtime error is reported.

The -noforms flag can only be used to run applications that do not require any forms. The application can contain forms, if you run the application in a way that does not call the forms. For example, you can start the application from a procedure instead of a frame by specifying the procname on the command line.

The -forms flag to call the forms system is the default and is included only for consistency.

SQL option flags

The SQL option flags are flags that affect the database behavior. Vision passes the flags to the database, which interprets them.

See the sql command description in the System Administrator's Guide for detailed information on these flags. The following SQL option flags are accepted when you run an image:

-f
+U
-l
-x

-Ggroupid

Lets you run or edit the application as a member of the group specified.

VMS: Capital letter flags require double quotes in VMS, for example: "**-**G*groupid*"

-R*rolename*

Runs the application image with the role specified.

If you specify a role name, you are prompted for the role's password. Roles are a feature of the Knowledge Management Extension.

VMS: Capital letter flags require double quotes, for example: "-R*rolename*"

-constants_file='*filename*'

Specifies a file containing values for the application's constants. If the -constants_file flag is specified, the values in the constants file override the values for the constants stored in the application.

The filename can be the full directory path name for the constants file.

-a *application_specific_parameters*

Allows the user to pass one or more application-specific parameters to the application.

The -a flag must be the last flag that appears on the command line. There must be a blank space between the -a flag and the first parameter that follows it.

Any characters following the -a flag are passed as a single string of parameters.

Retrieve the parameter values into the application by using the CommandLineParameters()function. See Using Application-specific Parameters (see page 279) for details.

Example: imagename command

To run the order entry application that you imaged as "orders," enter the following command at the operating system prompt:

Windows:

```
orders
```

The orders file must be in PATH.

UNIX:

```
orders
```

The orders file must be in PATH.

VMS:

```
runorders
```

or

```
run orders.exe
```

The application runs from the top frame or other default start frame (see Specify a Default Start Frame (see page 281)).

To access the application beginning with the frame AddOrders, the user enters:

Windows:

```
orders addorders
```

UNIX:

```
orders addorders
```

VMS: The imagename must be defined as a DCL foreign command. See the next section for instructions and an example.

As described in Creating a Command for an Imaged Application (see page 280), users can run the application with a command that you specify.

Using Application-specific Parameters

When users run an imaged application from the command line, you can let them specify parameter values that are passed back to the application itself. For example, you can ask the user to indicate a department name when starting the application. You then can use this value to restrict the records that the user can see on a Browse frame.

The -a flag indicates the start of a string of application-specific parameters. You retrieve the values into the application with the CommandLineParameters() function. Because of the way in which the CommandLineParameters() function retrieves the parameter values, the -a flag must be the last flag on the command line. You must leave a blank space between the -a flag and the first parameter that follows it.

You can write escape code (for example, Form-Start escape code for the top frame) to define in the application as many variables as you need to hold the values that the user specifies. You then can use these variables throughout the application, in any of the following ways:

- In escape code
- As parameters passed to a Vision frame
- As part of a visual query for an Append, Browse, or Update frame (for example, in a query restriction)

If you want to use multiple parameters, be aware that the CommandLineParameters() function always returns a single value—all the parameters concatenated into a single string, with the parameters separated by a single space (any multiple spaces that the user enters are compressed). You must write your own code for parsing this value into the separate variables for your application to use.

For an example of 4GL string-parsing code, and more information about how the CommandLineParameters() function retrieves parameter values, see the 4GL reference part of this guide.

Creating a Command for an Imaged Application

Create a command that includes any of the parameters listed above. When the user enters the command name to access the application, any of the parameters that you specify are invoked automatically.

Windows: Specify a command and runtime parameters in an icon. For example, let users use the icon "runorders" to run the application "order_entry" in the c:\usr\userdir directory with "neworders" as the start frame.

To do this, create an icon called "runorders" that contains this line:

```
c:\usr\userdir\order_entry neworders
```

UNIX: Specify a command and runtime parameters by writing a shell script. For example, you could let users enter the command "runorders" to run the application "order_entry" in the usr/userdir directory with "neworders" as the start frame.

To do this, create a file called "runorders" that contains this line:

```
usr/userdir/order_entry neworders
```

Verify that you have placed the UNIX shell script in the directory path so that all users of the application can access it. See your operating system documentation for more information about writing shell scripts.

The previous example assumes that the image name is the same as the application name. If you specify an image name that is different from the application name, use the image name in the icon or shell script.

VMS: Specify a command and runtime parameters by defining a foreign command. For example, you could let users enter the command "runorders" to run the application "order_entry" in the $DISK1:[USERDIR] directory with "neworders" as the start frame.

To do this, enter the following command at the operating-system prompt:

```
$ RUNORDERS=="$DISK1:[USERDIR]order_entry.exe neworders"
```

The above example assumes that the image name is the same as the application name. If you specify an image name that is different from the application name, use the image name in the foreign command.

See your VMS operating system documentation for more information about creating foreign commands.

After you create the command, the user simply types "runorders" at the operating-system prompt and accesses the application beginning with the "neworders" frame.

For a summary of the ways in which users can run an application, see Vision Applications from a User's Perspective (see page 297).

Specify a Default Start Frame

When you run a Vision application, Vision calls the top frame of the application flow diagram as the default start frame, unless you (or the user) specify a different frame. The following table describes the ways in which you can change the default start frame:

Method	Effect
Specify a different default start frame in the Application Defaults window. (You access this window by selecting MoreInfo in the Application Catalog window, then selecting Defaults.)	Vision uses the frame that you specify in the Application Defaults window as the start frame each time a user runs the application, unless you override it with one of the methods described below.
Create a command that contains a default start frame specification. (This method is described in Creating a Command for an Imaged Application (see page 280).)	The default start frame that you specify is called each time a user enters the command. You can specify commands with different default start frames for different users. This method gives you the most control over the way in which users access the application.

Method	Effect
Allow users to specify a default start frame when they call the application. (They do this by including the frame name with the executable image name.)	This method is the most flexible for users, but gives you the least control over the way in which users access the application.

Rename an Application

You can change the name of an application at any time, either while you are building it or after you are done. If you rename an application for which you already have created an image, you must create a new image. Also, remember to modify any command files that call the application.

To rename an application

1. Position the cursor on the application name in the Applications Catalog window.
2. Select Rename from the menu.

   Vision prompts you for the new name.
3. Enter a new name for the application.
4. Press Return.

   Vision renames the application.

Copying Applications into Different Databases

To copy an application from one database to another, use the copyapp command. The copyapp command copies information about an application as stored in the Ingres system catalogs. If the -s flag is specified, copyapp also copies the source code files for Custom frames and the application's compiled forms.

The copy process involves two steps:

- copyapp out

  This command transfers application data from the old database to an intermediate text file. Do not edit this file, because the application data is stored in a fixed order.
- copyapp in

  This command transfers application data from the text file to the new database.

The copyapp command notifies you if components in the new and old databases have the same names. By default, copyapp does not copy the component if there is a name conflict. If you use the -r flag, copyapp replaces the component in the new database with the same-named component from the old database.

Similarly, you can replace an existing application by using the -r flag. By default, the copyapp in operation does not copy the application if an application by the same name exists in the database.

The copyapp command does not transfer database tables, or other database components, such as rules. In order for your application to work correctly, you must be sure that the database contains the correct components.

You cannot use the copyapp command to merge applications. Use the iiexport and iiimport commands to copy components from one application to another.

You can also copy individual components of an application—such as forms and reports—from one database to another using copyform, copyrep, and similar utilities. To copy database tables, you can use the copydb and unloaddb utilities. See the *Database Administrator Guide* for more information about any of these utilities.

The following sections describe the two steps of the copyapp operation.

copyapp out Command—Copy an Application Out of a Database

To copy an application out of its original database, issue the following command at the operating-system prompt.

This command has the following format:

```
copyapp out [-ddirname][-tfilename] [-uusername]
          [-lfilename] olddbname applname
```

-ddirname

Specifies a directory in which to place the intermediate file.

The default is the current directory.

-tfilename

Indicates a name for the intermediate file.

If the flag is not set, the intermediate file is named *iicopyapp.tmp.*

-u*username*

Identifies you to Ingres as the user with login name *username*.

To use this option, you must be a privileged user.

If you are using Enterprise Access products, refer to your Enterprise Access documentation before using this parameter.

olddbname

The name of the database in which the application currently resides

applname

The name of the application

-l*filename*

Creates a file containing a list of the names of source files for Custom frames only

Copying Applications to a Different Location

You can use copyapp to copy an application to a database in a different Ingres location than the current database with copyapp, with the following exception:

You cannot copy an application from a database on a remote host to a new location on a different remote host.

If you are copying an application to a database in a different Ingres location than the current database, you must:

- Use -s or -a to copy an application with Vision (not custom) frames. Otherwise, all frames are marked as custom.
- Change the source code directory path in the Application Defaults window, if you did not use the -s or -a flags
- Change any hard-coded path names that appear in the 4GL code in Custom frames

copyapp in Command—Copy an Application into a Database

To copy an application into a database, use the copyapp in command.

This command has the following format:

```
copyapp in [-c] [-ddirname] [-nnewapplname] [-p] [-q] [-r]
        [-s[dirname ] |-a[dirname ] ] [-uusername]
        newdbname intfilename [-lfilename]
```

-c

Deletes the intermediate file

-d*dirname*

Indicates the directory that you have specified for the intermediate file

Provide the full directory pathname. The default is the current directory.

-n*newapplname*

Lets you specify a new name for the application in the new database. The default is the same name as in the old database.

-p

Suppress messages about name conflicts. The default is to display messages.

-q

Causes copyapp to quit if there is any name conflict.

When this flag is set, no changes are made to the database if a name conflict is found.

If you specify the -q flag, copyapp is performed as a single transaction. The copyapp transaction locks system catalogs; for this reason, you must not specify -q when users are connected to the database. In addition, the transaction is logged in the log file; you must be sure that the log file is large enough to accommodate the copyapp transaction.

If neither -q or -r is specified and a duplicate name is encountered, the copy is not done and terminates with an error message. In this case, the application can exist in the new database in a partially copied state.

-r

Overwrites components with the same name.

If neither -q or -r is specified and a duplicate name is encountered, the copy is not done and terminates with an error message. In this case, the application can exist in the new database in a partially copied state.

-u*username*

Identifies you to Ingres as the user with login name *username,* and transfers ownership of the application and its components to that user.

To use this option, you must be a privileged user.

If you are using Enterprise Access products, refer to your Enterprise Access documentation before using this parameter.

-s[*dirname]*

Specifies a new directory for source files. If a directory name is not specified, the current directory is used.

For Custom frames, copies 4GL source files.

For *non-custom* Vision frames, copyapp transfers no files. The frames are marked as *new*. The source files for these frames are regenerated at the next Image or Go operation.

Do not use this flag with the -a flag.

-a[*dirname*]

Specifies a new source directory name for the application, but does not transfer source files. If a directory name is not specified, the current directory is used.

This flag marks Vision (non-custom) frames as *new.* The source file is regenerated at the first Image or Go operation.

Do not use this flag with the -s flag.

newdbname

Specifies the name of the database into which the application is to be copied

intfilename

Specifies the name of the intermediate file

-l*filename*

Creates a file containing a list of the names of source files for Custom frames only.

If -s is specified, the source files in this list (Custom frames) were copied.

If -a is specified, the source files in this list (Custom frames) were not copied.

Example: copyapp out and copyapp in commands

As an example of the copyapp command, assume that you want to move the Sales application from the Accounts database into the Newdb database. The following statement performs the first part of the task, copying the application into an intermediate text file called *sales.tmp*:

```
copyapp out -tsales.tmp accounts sales
```

After copyapp copies the application into the *sales.tmp* file, you can copy *sales.tmp* into the Newdb database by executing the following command:

```
copyapp in -a newdb sales.tmp
```

Copy the Vision "new_emp" application from the "employee" database to the "employee2" database. Use the default intermediate text file, and use the current working directory as the new application's source directory.

```
copyapp out employee new_emp
copyapp in -a employee2 iicopyapp.tmp
```

Copying Application Components

You can copy components of an application to another application, called the target application, using the iiexport and iiimport commands. Unlike the copyapp command, which copies an entire application, iiimport, and iiexport copy individual components.

The components you can copy are frames, procedures, records, global variables, and global constants. You can copy all the components of a particular type using one of the -all flags:
-allframes, -allprocs, -allrecords, -allglobals, -allconstants.

Copying components requires two steps:

1. Use iiexport to extract the components to an intermediate file.
2. Use iiimport to load the components from the intermediate file into the target application.

By default, the intermediate file is called iiexport.tmp. You can change the name of the intermediate file with the -intfile flag. If the applications are not on the same machine or virtual file system, you must copy the intermediate file to the target machine before running iiimport.

Using the iiimport and iiexport commands allows you to merge components into an existing application. For example, if you are working on a development team, the team can have one master application. Each team member can develop frames and other components independently, and use iiimport and iiexport to copy the components into the master application.

You could also use the intermediate file created by iiexport as a snapshot of a frame or other component at a particular stage of development. For example, you can check the file into a source control system or store it as a backup file.

See iiexport Command—Copy Application Components to a File (see page 289) and iiimport Command—Import Application Components (see page 291) for detailed syntax for these commands.

How Dependencies Are Handled

This section describes how iiimport and iiexport handle the relationships between frames and dependent components when you copy a frame.

A frame in a Vision application is interconnected with other frames in the application. For example, a frame can be a parent frame that calls one or more child frames. The parent/child structure of the application is displayed in the application flow diagram. Other components of an application, for example menu items and parameters, depend on the relationships of one frame to another.

Because the same frame can appear more than once in the application, a frame can have multiple relationships in the application. For example, a frame can have one child frame in one place, and another child frame in another place.

When you are copying a frame from one application to another, the iiimport and iiexport commands do not know the relationships that can exist between that frame and others in the target application.

The parent/child structure of the target application can be different from the structure of the exporting application. For the frame to be consistent and usable after the frame is imported, the dependent components, such as menu items, must reflect the structure of the target application. For this reason, components which depend on the relative positions of frames to one another are not included as part of the frame definition when you copy a frame.

The iiimport command does not change the following dependent components:

- Child frames (unless specified explicitly)

  If the frame is a parent frame in a Vision application, iiimport does not copy the child frames unless you explicitly specified the child frames to iiexport.

  If parent and child frames are both copied (specified explicitly to iiexport), the frames are copied singly, and not in relationship to each other. The iiimport command does not change the parent/child structure of the target application, or the related menu items.

- Menu items associated with child frames

  Because the menu for a frame reflects its child frames, iiimport preserves the menu for the structure of the target application, rather than changing it to reflect the structure in the exported application.

  Any user-defined menu items *are* copied as part of the frame, because they are contained in escape code and do not depend on the parent/child structure.

- Frame parameter passing

  The iiimport command preserves the passing of frame parameters you have specified in the target application. If the exported application passes parameters between frames, this information is not copied when the frame is copied. You must specify this in the target application. (Any local variables that you defined to use for passing parameters *are* copied.)

- Global components (unless specified explicitly)

  A frame can also be dependent on global components, that is, global constants, record types, global procedures, or global variables. For the frame to run correctly, those components must exist in the target application. When you copy a frame with iiexport and iiimport, global components are not automatically copied, even if the frame references them. To copy global components, you can specify them explicitly to iiexport.

iiexport Command—Copy Application Components to a File

The iiexport command copies one or more application components to a file. A component can be a frame, procedure, record type, global variable, or a global constant. You can copy all the components of a particular type from an application using one of the -all flags: -allframes, -allprocs, -allrecords, -allglobals, -allconstants.

When you copy a frame, iiexport copies the frame definition, including local components and Escape code. However, iiexport does not copy global components or child frames, unless they are explicitly specified. See How Dependencies Are Handled (see page 288) for more information.

This command has the following syntax:

```
iiexport dbname appname [-intfile=filename] [-listing=filename]
     [-user=username] component=name{,name} {component=name{,name}}
```

component

Is one of the following: -frame, -proc, -record, -global, -constant, -allframes, -allprocs, -allrecords, -allglobals, or -allconstants

name{,name}

Is a comma-separated list with no spaces

dbname

Specifies the database from which the component is being copied

appname

Specifies the application from which the component is being copied

-intfile=*filename*

Speciries the intermediate file created by iiexport. The *filename* can be the full pathname of the file. If you specify the -intfile flag, you must indicate a file name. If you do not specify the -intfile flag, iiexport names the intermediate file iiexport.tmp.

-listing=*filename*

Specifies a file that lists the names of the source files for each frame copied. If you specify the -listing flag, you must indicate a file name. If -listing is not specified, Vision does not create a listing file.

-user=*username*

Runs the command as the user specified by *username.*

To use this option, you must be a privileged user.

If you are using Enterprise Access products, refer to your Enterprise Access documentation before using this parameter.

-frame=*name*{,*name*}

Indicates the name of one or more frames to be copied

-proc=*name*{,*name*}

Indicates the name of one or more procedures to be copied

-record=*name*{,*name*}

Indicates the name of one or more record types to be copied

-global=*name*{,*name*}

Indicates the name of one or more global variables to be copied

-constant=*name*{,*name*}

Indicates the name of one or more global constants to be copied

-allframes

Indicates that all frames must be copied from the application

-allprocs

Indicates that all procedures must be copied from the application

-allrecords

Indicates that all record types must be copied from the application

-allglobals

Indicates that all global variables must be copied from the application

-allconstants

Indicates that all global constants must be copied from the application

iiimport Command—Import Application Components

The iiimport command copies application components from an intermediate file into an existing application. The intermediate file, created by iiexport, contains the definitions for the components you specified when you ran iiexport. See the description of iiexport for the types of components you can copy.

When you import a component, the iiimport command assumes that the component exists in the application, and revises the existing version of the component. When you import a frame, the existing frame definition, including local components and Escape code, is replaced with the new definition. The iiimport command does not change menu items for a frame, except for user-defined menu items included with Escape code. See How Dependencies Are Handled (see page 288) for information on how iiimport handles menu items and other dependent components.

You can check whether a component exists in the target application by using the -check option. Running iiimport with the -check flag does not actually copy or replace any components, but reports on the status of the components in the target application.

If a Vision frame does not exist in the target application, iiimport copies the frame but does not incorporate the frame into the Application Flow Diagram. From the Application Flow Diagram, you can use the Insert option to include the frame at the desired location.

You can import a component that does not exist in the target application, but the target application itself must exist. To copy an entire application into a database, use the copyapp utility.

All components copied in are assigned to the owner of the target application.

This command has the following format:

```
iiimport dbname appname [-intfile=filename] [-listing=filename]
     [-user=username] [-check | -copysrc] [-dropfile]
```

dbname

Specifies the database from which the component is being copied

appname

Specifies the application from which the component is being copied

-intfile=*filename*

Specifies the intermediate file created by iiexport. The *filename* can be the full pathname of the file. If you specify the -intfile flag, you must indicate a file name. If you do not specify the -intfile flag, iiexport names the intermediate file iiexport.tmp.

-listing=*filename*

Specifies a file that lists the names of the source files for each frame copied. If you specify the -listing flag, you must indicate a file name. If -listing is not specified, Vision does not create a listing file.

-user=*username*

Runs the command as the user specified by *username.*

To use this option, you must be a privileged user.

If you are using Enterprise Access products, refer to your Enterprise Access documentation before using this parameter.

-frame=*name*{,*name*}

Indicates the name of one or more frames to be copied

-proc=*name*{,*name*}

Indicates the name of one or more procedures to be copied

-record=*name*{,*name*}

Indicates the name of one or more record types to be copied

-global=*name*{,*name*}

Indicates the name of one or more global variables to be copied

-constant=*name*{,*name*}

Indicates the name of one or more global constants to be copied

-allframes

Indicates that all frames must be copied from the application

-allprocs

Indicates that all procedures must be copied from the application

-allrecords

Indicates that all record types must be copied from the application

-allglobals

Indicates that all global variables must be copied from the application

-allconstants

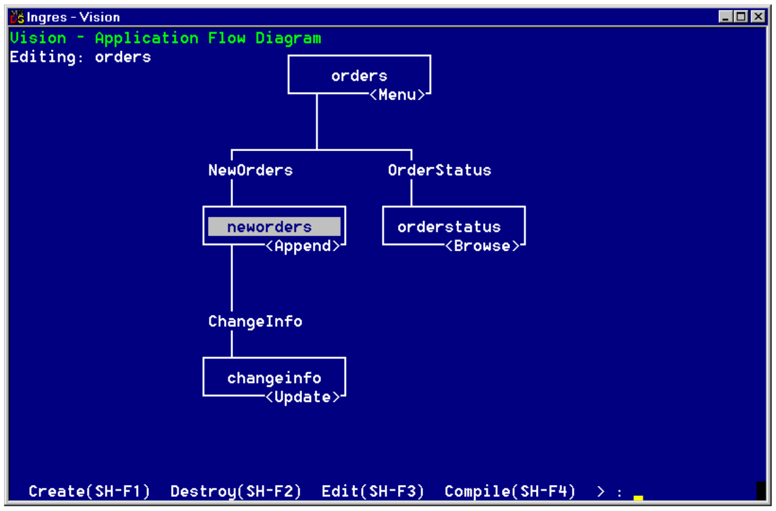
Indicates that all global constants must be copied from the application

Master Application Scenario

If multiple developers are working on an application, they can develop portions of the application independently. Then they can use iiimport and iiexport to copy frames and other components into a master application as they are developed.

In this example, the master application is the "orders" application. The development team designs the overall structure of the application with the application flow diagram.

The application flow diagram for the "orders" application is shown in the following figure:

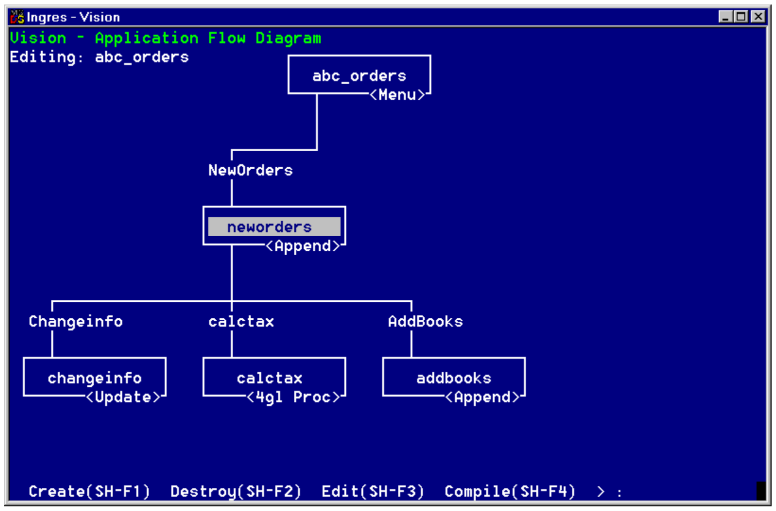


Now the developers are assigned to different pieces of the application. Developer A is assigned to modify "neworders," an Append frame called by the "orders" Menu frame.

Developer A makes the following changes to the application:

- Modifies the "neworders" frame
- Adds the called procedure "calctax"
- Adds the child frame "addbooks"
- Passes some parameters from neworders to changeinfo
- Passes parameters from neworders to "addbooks"
- Adds escape code to "changeinfo"

The application flow diagram in Developer A's version of the application looks like the one in the following figure:



Examples: Master Application Scenario

The following examples use the scenario described in the Master Application Scenario section. Developer A wants to merge the revised frames and other components into the master "orders" application.

- Replace the master "neworders" frame with the "neworders" frame in abc_orders:

  ```
  iiexport tutor abc_orders -frame=neworders
  iiimport tutor orders
  ```

  After the import, the child frames "calctax" and "addbooks" do not appear in the "orders" application, or in the menu for the "neworders" frame in the target application. The "changeinfo" frame is not changed.

- Copy the procedure "calctax" into the master application:

  ```
  iiexport tutor abc_orders -proc=calctax
  iiimport tutor orders
  ```

  If the procedure uses a global variable *taxamt*, the variable must exist in the target application. If it does not, copy the variable by specifying it to iiexport:

  ```
  iiexport tutor abc_orders -global=taxamt
  iiimport tutor orders
  ```

  You can also copy the procedure and the variable at the same time:

  ```
  iiexport tutor abc_orders -proc=calctax -global=taxamt
  iiimport tutor orders
  ```

- Copy the frame "neworders" and its child frame, "changeinfo"

  ```
  iiexport tutor abc_orders -frame=neworders,changeinfo
  iiimport tutor orders
  ```

  In this case, because the child frame "changeinfo" is specified to iiexport, the "changeinfo" frame in the target application is changed. However, the parameters passed from "neworders" to "changeinfo" are not changed.

- Copy the frame "neworders" and its child frame "addbooks"

  ```
  iiexport tutor abc_orders -frame=neworders,addbooks
  iiimport tutor orders
  ```

  In this case, the child frame "addbooks" is specified to iiexport, so the frame is copied. However, the "addbooks" frame does not exist in the target application's flow diagram. The iiimport operation does not know the relation of the frame to the "neworders" frame. The "addbooks" frame is copied into the "orders" application catalog. To link the frame to the application flow diagram, choose Insert from the menu. Any parameters passing from the "neworders" frame to the "addbooks" frame must be specified in the "orders" application.

Destroy Applications

You can destroy an application that you are no longer using. When you destroy an application, Vision destroys all the frames in the application. However, Vision does not destroy the forms and source code files. At the operating system level, delete the source code files by using the appropriate command for your operating system.

To destroy an application

1. Position the cursor on the application name in the Applications Catalog window.
2. Select Destroy from the menu.

   Vision asks you to confirm that you want to destroy the application.
3. Enter y and press Return.

   Vision destroys the application.

After you destroy an application, you can run Vision's Cleanup utility as described in Clean Up a Database (see page 255).

Appendix A: Vision Applications from a User's Perspective

This section contains the following topics:

This appendix helps you understand how to develop your applications so that users can run them and perform operations in specific ways. This appendix describes:

- How users can employ the ListChoices operation, so that you can decide when to include Lookup tables in your visual queries
- The menu operations that Vision generates to permit specific user actions

  Vision Append, Browse, and Update frames permit a large combination of user operations. This section helps you specify visual queries so that your frames perform, as you want them to.
- How users can run an application

By comparing the methods available, you can control user access while letting users run the application more easily and quickly.

For instructions on how to specify the functions described in this appendix, see the appropriate chapters of this guide.

Using Lookup Tables and the ListChoices Operation

You can include Lookup tables in the visual queries for Append, Browse and Update frames to:

- Let users choose an item from a selection list to insert values onto a form
- Validate an entry that a user types directly into a field
- Have Vision enter values directly into fields on a form

Vision generates a ListChoices menu item that lets users activate these various functions.

This section describes each of these functions and ends with discussions of Lookup tables and Listchoices.

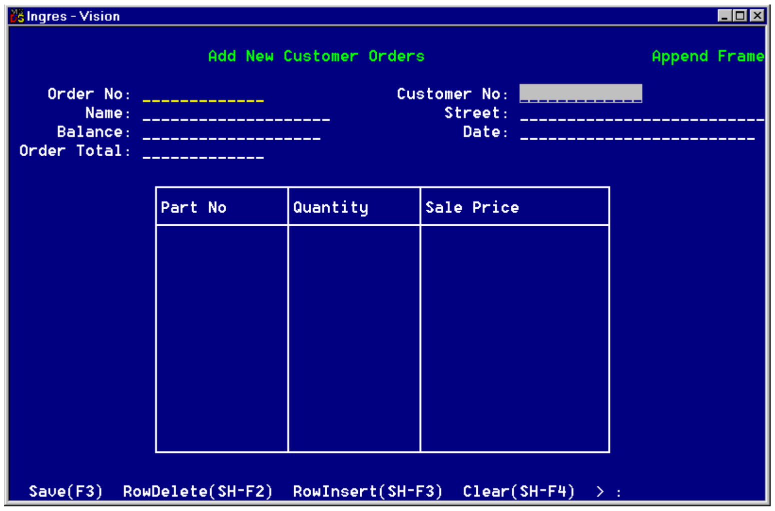
Choosing from a Selection List

You can include a Lookup table on a visual query for:

- An Append frame
- A Browse or Update frame that lets users enter query qualifications and select Go to retrieve particular records

Then users can select the ListChoices operation to display a selection list. You specify in the visual query which columns of the Lookup table to display in the selection list.

For example, the AddOrders frame in the following figure lets users add customer orders. A customer number is required for each order, but a user cannot know this number. You can include the Customers table as a Lookup table to display a selection list that contains the customer name for each customer number.



```
                    Add New Customer Orders                        Append Frame

     Order No: _____________                Customer No: _____________
         Name: ____________________              Street: ___________________________
      Balance: ___________________                 Date: _________________________
  Order Total: _____________

                  Part No       Quantity       Sale Price

  Save(F3)  RowDelete(SH-F2)  RowInsert(SH-F3)  Clear(SH-F4)  > :
```

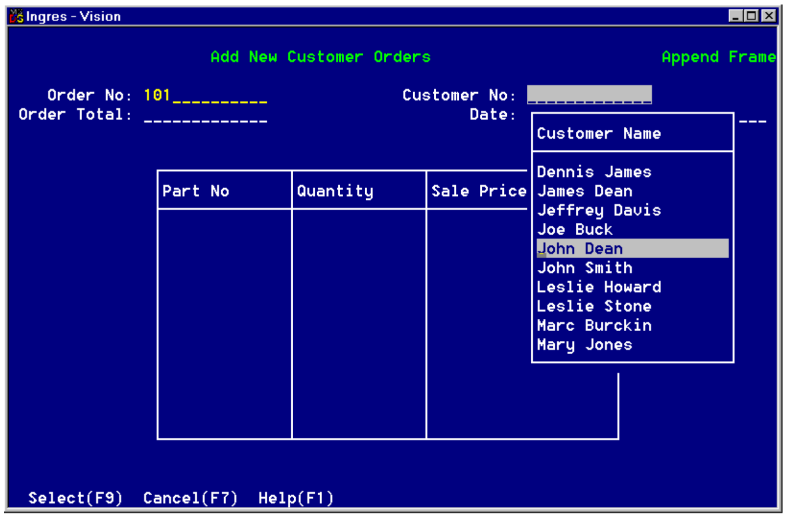
The Lookup table is activated on the customer number field.

When a user calls the AddOrders frame, the Lookup table functions as follows:

User Action	Vision Response
Selects ListChoices with the cursor on the customer number field	Displays the selection list containing customer names (see the following figure)
Then chooses the desired name from the selection list	Enters the appropriate customer number onto the form (see the second figure following)

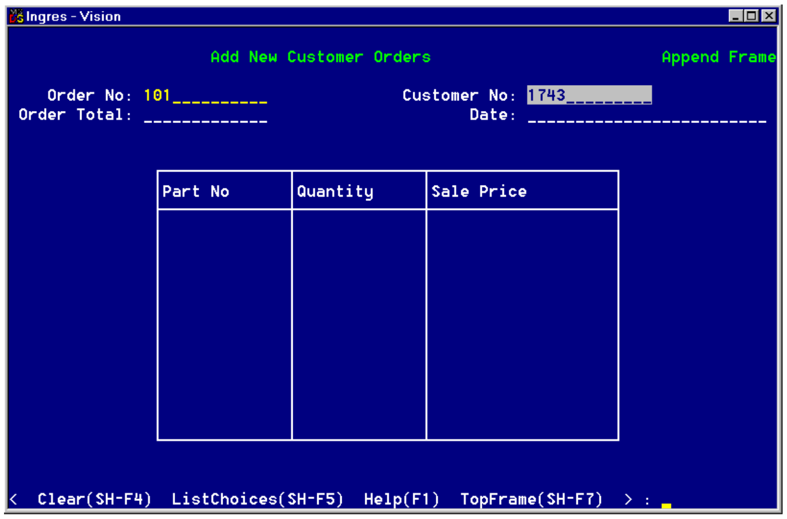
In this example, the customer number is not actually displayed in the selection list. Vision automatically retrieves the number that corresponds to the customer name that the user selects. By using a selection list in this way, users can enter unknown values by selecting known ones.

The following figure illustrates choosing a name form the selection list:



```
Ingres - Vision
                    Add New Customer Orders                    Append Frame

    Order No: 101__________              Customer No: _____________
 Order Total: _____________                     Date:  ┌────────────────────┐ ___
                                                       │Customer Name       │
                                                       ├────────────────────┤
               ┌─────────────┬─────────────┬──────────│Dennis James        │
               │Part No      │Quantity     │Sale Price│James Dean          │
               ├─────────────┼─────────────┼──────────│Jeffrey Davis       │
               │             │             │          │Joe Buck            │
               │             │             │          │John Dean           │
               │             │             │          │John Smith          │
               │             │             │          │Leslie Howard       │
               │             │             │          │Leslie Stone        │
               │             │             │          │Marc Burckin        │
               │             │             │          │Mary Jones          │
               │             │             │          └────────────────────┘
               │             │             │                   │
               └─────────────┴─────────────┴───────────────────┘

  Select(F9)  Cancel(F7)  Help(F1)
```

The following figure illustrates Vision entering the customer number on the form:



```
Ingres - Vision
                    Add New Customer Orders                    Append Frame

    Order No: 101__________              Customer No: 1743_________
 Order Total: _____________                     Date: _________________________

               ┌─────────────┬─────────────┬───────────────────┐
               │Part No      │Quantity     │Sale Price         │
               ├─────────────┼─────────────┼───────────────────┤
               │             │             │                   │
               │             │             │                   │
               └─────────────┴─────────────┴───────────────────┘

<  Clear(SH-F4)  ListChoices(SH-F5)  Help(F1)  TopFrame(SH-F7)  > :
```

Validating User Entries

You can include a Lookup table for a field on a frame. The Lookup table validates the user's entry when the user enters a value directly into a field, rather than selecting ListChoices and choosing from the selection list.

For example, assume that a user calls the AddOrders frame described above (see the first figure in this appendix). However, the user enters a value into the customer number field rather than using ListChoices to choose from the selection list.

Here is how the Lookup table functions in this case:

User Action	Vision Response
Enters a customer number	Checks the Customers table for that number If the customer number is in the table, moves the cursor to the next field. If the customer number is not in the Lookup table, displays an error message to the user.
Then either enters another customer number, or	Checks the entry again as above
Uses ListChoices with the selection list as described in the previous example	Enters the customer number on the form as in the previous example

Entering Lookup Table Values on a Form

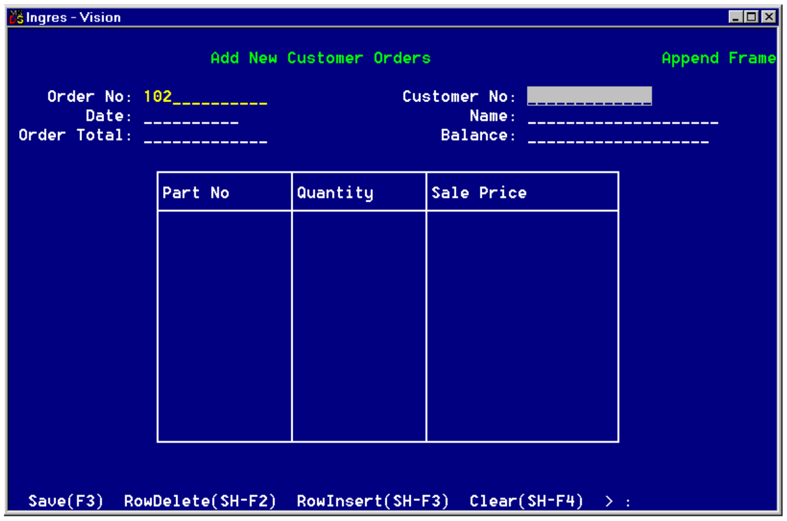
Vision can enter the value of columns from a Lookup table into fields on a form. These values can be in addition to the column that activates the Lookup table. Vision enters these values whether or not a user chooses from the selection list.

Vision enters values for any Lookup table columns that you display as fields on the form. These values are display-only; users cannot change them.

As an example, assume that you want to display not only the customer number, but also the customer's name and account balance (the Balance column of the Customers table) on the AddOrders frame illustrated above (see the first figure in this appendix).

To do this, indicate in the visual query that the name and balance columns must be displayed as fields on the form. Then edit the form to move these fields to an appropriate location on the form.

The following figure shows how the form now appears to the user:

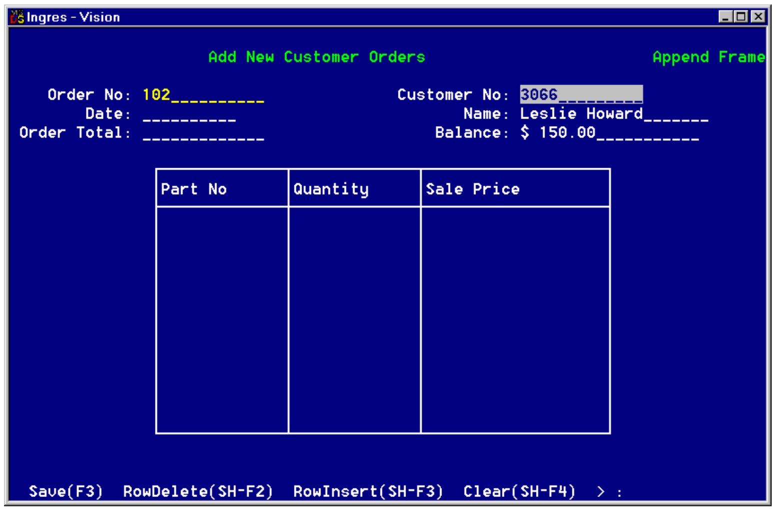


```
Ingres - Vision

                 Add New Customer Orders                         Append Frame

   Order No: 102__________             Customer No: _____________
       Date: __________                       Name: ____________________
Order Total: _____________                 Balance: ___________________

                Part No       | Quantity     | Sale Price

 Save(F3)  RowDelete(SH-F2)  RowInsert(SH-F3)  Clear(SH-F4)  > :
```

When the user calls the AddOrders frame, the Lookup table now functions as follows:

User Action	**Vision Response**
Selects ListChoices with the cursor on the customer number field	Displays the selection list containing the customer names
Chooses the desired name from the selection list	Enters the appropriate customer number onto the form. Also displays the customer name and account balance (see the following figure)

The following figure illustrates Vision entering the name and balance on the form:



```
Ingres - Vision

                 Add New Customer Orders                    Append Frame

   Order No: 102__________            Customer No: 3066_________
       Date: __________                      Name: Leslie Howard_______
Order Total: _____________                Balance: $ 150.00___________

                +-------------+-------------+--------------------+
                | Part No     | Quantity    | Sale Price         |
                +-------------+-------------+--------------------+

  Save(F3)  RowDelete(SH-F2)  RowInsert(SH-F3)  Clear(SH-F4)  > :
```

You can display a field on the form without having to display it as a column of the selection list. In this example, the Balance field is not displayed in the selection list.

Using a Lookup Table Without a ListChoices Menu Item

There is one case in which you can include a Lookup table without Vision generating a ListChoices menu item or making a selection list available. This occurs if you create a Browse or Update frame and:

- Include a Lookup table in the visual query
- Display columns of the Lookup table as fields on the form
- Do not allow users to enter a qualification on the form

When a user calls the frame, the user cannot enter a qualification and select Go to retrieve a record. Vision:

- Displays the form with values already entered
- Retrieves data into the form based on the values of the displayed columns of the Lookup table
- Does not generate a ListChoices menu item for choosing from a selection list, because the user has no control over retrieving records onto the form

Using ListChoices Without a Lookup Table

Vision generates a ListChoices menu item for each Append, Browse and Update frame (except as described immediately above). If you do not include a Lookup table in a frame definition, the ListChoices operation functions as follows:

- If you use the Visual-Forms-Editor to specify a validation for any fields on the form, Vision checks this validation when a user selects ListChoices.

  For example, you can specify a validation to make sure that the customer number that a user enters is a current value from the Customers table. See *Character-based Querying and Reporting Tools User Guide* for details on VIFRED validations.

- If you do not specify a validation, Vision displays a message informing the user that ListChoices is not a valid operation for the field.

Allowing Specific User Operations

You can use Append, Browse and Update frames in your Vision applications to let users perform a wide variety of operations. Your visual query specifications determine the menu items that Vision generates for users to perform these operations.

The following tables summarize the available operations for Append, Browse, and Update frames.

User Operations on Append frames

Append frames let users add new records to Master and Detail tables. The following table describes the operations that users can perform on Append frames and the corresponding menu items that Vision generates:

User Operation	Menu Item that Vision Generates	Visual Query Specification
Add new records to Master and Detail tables and save them to the database	Save	By default
Open a blank row in the Detail table, to add new data	RowInsert	By default, if the frame definition includes a Detail table
Delete a new row in the Detail table, rather than	RowDelete	By default, if the frame definition includes a Detail

User Operation	Menu Item that Vision Generates	Visual Query Specification
saving it to the database		table

Append frames also contain the menu items described in Additional Generated Menu Operations (see page 305).

User Operations on Browse frames

Browse frames let users view data retrieved from the database. There are various ways to control the data that is displayed on a Browse frame.

The following table describes the operations that users can perform on Browse frames and the corresponding menu items that Vision generates:

User Operation	Menu Item that Vision Generates	Visual Query Specification
Enter a qualification to retrieve particular records from the database	Go	Set the Qualification Processing frame behavior to "enabled"
If you allow user query qualifications, clear the display to enter another qualification	Clear	By default, if you allow user query qualifications
Display the next Master table row (and any corresponding Detail table rows)	Next	Set the Next Master Menu Item frame behavior to "enabled"

Browse frames also contain the menu items described in this appendix in Additional Generated Menu Operations (see page 305).

If you do not allow users to enter query qualifications, Vision does not generate a ListChoices menu item.

User Operations on Update frames

Update frames let users view data retrieved from the database. In addition, users can change this data and, if you permit, delete data or add new data. You can control the display of data in the same ways as on Browse frames.

The following table describes the operations users can perform on Update frames—in addition to those for Browse frames—and the corresponding menu items that Vision generates:

User Operation	Menu Item that Vision Generates	Visual Query Specification
Write a changed record to the database (or confirm a deletion of a record)	Save	Included in the default frame definition for Update frames
Change a record, then save it as a new record without affecting the original data	AddNew	Set the value of the Insert field to "y" for each table to which users can add records
Add new records to the Master or Detail table	AppendMode	By default, if you allow insertions to the Master or Detail table as described above
Open a blank row in the detail table on which to add a new record	RowInsert	By default, if you allow insertions to the Detail table as described above
Delete records from the Master table and/or the Detail table	Delete	For details on allowing users to delete records on Update frames, see Defining Frames with Visual Queries (see page 117).
Delete the row in the Detail table on which the cursor is positioned	RowDelete	Allow Detail table deletions are described in Defining Frames with Visual Queries (see page 117).

Additional Generated Menu Operations

In addition to the menu operations to manipulate data, all Vision-generated frames include menu operations to let users perform standard operations. Even though Menu frames do not have visual queries, they contain these standard menu items too.

Standard Operations and Menu Items

The following tables lists the standard operations that users can perform on Menu, Append, Browse, and Update frames and the corresponding menu items that Vision generates.

User Operation	Menu Item that Vision Generates
Display information that Vision generates about a frame You can edit the text for these windows as described in the Vision-Generated Help Files section in Modifying Vision Code (see page 227).	Help
Return to the previous frame; on submenus, return to the previous menu	End (on all frames except the top frame of an application)
Call the top frame from any point in the application below the top frame	TopFrame (on all frames except the top frame of an application)
Leave the application	Quit (on the top frame only)

ListChoices and Table Field Menu Operations

Vision generates a ListChoices menu item for each Append, Browse or Update frame. The Using Lookup Tables and the ListChoices Operation section above discuss how users can use this operation.

The following table lists the menu items that Vision generates when a user displays a selection list with the ListChoices operation. These operations also are available on menus that are displayed as table fields:

User Operation	Menu Item that Vision Generates
Select an item from the list	Select
Return to the previous frame without making a selection	Cancel
Display information about the current frame	Help

Table Field Operations

The following operations are available on any Vision-generated frames that contain table fields. These include Append, Browse, and Update frames in which the Master table or Detail table is displayed as a table field, as well as Menu frames displayed as table fields.

User Operation	**Menu Item**
Enter a qualification to find a particular record in the table field	Find
Move to the first retrieved row of the table field	Top
Moves to the last retrieved row of the table field	Bottom

Vision does not generate menu items for these operations; users can activate them using the keyboard function keys to which they are mapped. To let users view the current key mappings on any frame, Vision generates a Keys menu item on the submenu of the Help operation.

Running an Application

There are several ways in which you can let users run an application. In general, these various methods differ by:

- Simplicity of access for users
- Control of user access by the developer
- Complexity of specification by the developer

The following are the methods by which users can run an application. They are presented in the order in which they satisfy the above criteria:

- In Vision, from the Applications Catalog window
- From the operating system, with the vision command
- By specifying an image file name
- By running a command

This section describes each method of running the application and the implications of its use.

Using the Vision Applications Catalog

You can run a completed or partial application with the Go menu operation on the Vision Applications Catalog window. You generally use this method to test applications during development or to run simple applications for your own use.

Do not let end users in a production environment run an application because they gain access to the Vision development environment and are not familiar with Vision windows and operations.

Using the Vision Command

You can run an application through Vision directly from the operating system prompt. To do this, include an application name and frame name in the vision command specification, using the following syntax:

```
vision [nodename::]dbname applicationname framename
```

For example, to run an application called "order_entry" in the "inventory" database from the "addorders" frame, enter:

```
vision [hq::]inventory order_entry addorders
```

When you include the frame name, Vision assumes that you want to run the application rather than edit it. Therefore, you must include the name of a frame, even if this frame is the top frame.

When you exit an application that you run in this way, the system returns to the operating system prompt. Therefore, users do not have access to the Vision development environment.

However, this method is not advisable for end users in a production environment because:

- They must enter a lengthy command specification.
- They must know the command syntax and the names of all the components.
- They can experience a delay in starting the application, because the system still must call Vision first.

For more information about the vision command, see Starting Vision or ABF with Command Line Options (see page 1381).

Using an Image File Name

You can create an executable version of an application that users can run without going through Vision. You do this by creating an image of the application.

The image file provides Vision with the application name and database name that it needs to run the application. You can give the image a name that is easier for users to enter than the application name.

For example, you can create an image called "orders" for the "order_entry" application. To run the application, a user types at the operating system prompt:

Windows:

```
orders
```

UNIX:

```
orders
```

VMS:

```
run orders
```

By default, Vision runs the application from the top frame or other default start frame. To start the application with a different frame, a user enters the frame name after the image name.

For example, to start the application in the example above from the NewOrders frame, a user enters:

Windows:

```
orders neworders
```

UNIX:

```
orders neworders
```

VMS: The *imagename* must be defined as a DCL foreign command. See Creating a Command for an Imaged Application (see page 280) for instructions and an example.

Creating an image can take time. However, running an imaged application is more efficient than going through Vision because:

- You can create an image name that is easy for users to remember and enter.
- The application must start more quickly, because users runs an executable file directly from the operating-system level.
- Users do not have access to the Vision development environment.

However, users still must enter any optional parameters, such as a specific start frame. Therefore:

- Users must know the parameter names and syntax
- You do not have total control over how a user accesses the application

Using a Command to Run an Application

After you create an executable image of an application, you can use it in a command that users enter to run the application. A command is an executable file that includes the image file name and any parameters that you specify. A command can be one of the following, depending on your system:

Windows:

`icon`

UNIX:

`shell script`

VMS:

DCL foreign command

Creating a command requires, on your part, the most planning of all the access methods. However, using a command is most desirable in a full production environment, especially if many users with different needs are accessing the application.

Using a command provides these advantages over the other methods of running an application:

- Users can start the application quickly and easily.
- Users only need to know a single command to run the application, without having to remember syntax and component names.
- You can specify commands to control precisely how users access the application.
- You can create different commands for different users based on their functions, information needs, security levels or other factors.

For example, various users can access the order entry application. You can create various commands to:

- Let some users call a Browse frame to view customer information
- Allow clerks to call frames to perform such functions as adding new orders and updating customer records
- Let managers call the application to access confidential financial information

Each of these commands calls the application with a different start frame. You can create as many different commands as you need.

For more information about creating image files and commands, see imagename Command—Run a Completed Application (see page 275).

Appendix B: Accessing Vision through ABF

This section contains the following topics:

This appendix describes the development of Vision frames and procedures using ABF. It also describes how to access Vision functions and procedures through ABF.

A brief, introductory description of the ABF Edit an Application window is provided. For complete information on the Edit an Application window, in the ABF part of this guide, see Building Applications (see page 351).

Edit an Application Window

If you are familiar with ABF, you can work with Vision so that it appears more like ABF by accessing the ABF environment from Vision through the Edit an Application window.

The Edit an Application window provides a different way of looking at the frames in an application from the application flow diagram. The primary differences are:

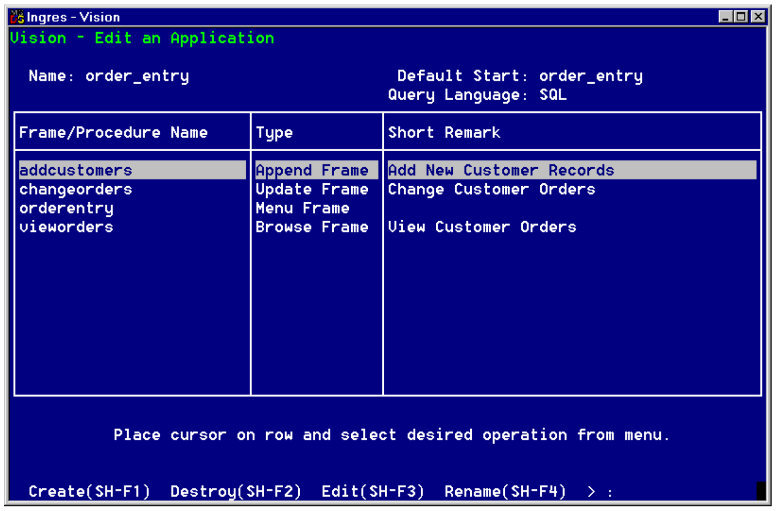
- The Edit an Application window simply lists the frames in the application, rather than showing relationships between them.

  For large applications, this can be a more convenient way to select a particular frame than in the application flow diagram.

- The Edit an Application window lists all frames in the application, including those that you have removed. Only those that you have destroyed are not available.
- The application flow diagram does not include frames that you call from escape code or from frames that Vision has not generated. You must access these called frames through the Edit an Application window.

To access the Edit an Application window from Vision

1. Select any frame as the current frame in the application flow diagram.
2. Select Catalog from the menu. Vision displays the Edit an Application (Edit an Application) window, as shown in the following figure.



```
Vision - Edit an Application

  Name: order_entry                          Default Start: order_entry
                                            Query Language: SQL

 Frame/Procedure Name     | Type          | Short Remark
 addcustomers             | Append Frame  | Add New Customer Records
 changeorders             | Update Frame  | Change Customer Orders
 orderentry               | Menu Frame    |
 vieworders               | Browse Frame  | View Customer Orders

           Place cursor on row and select desired operation from menu.

  Create(SH-F1)  Destroy(SH-F2)  Edit(SH-F3)  Rename(SH-F4)  > :
```

Create Frames and Procedures through ABF

Use the Application Flow Diagram Editor to create frames for which Vision generates the code—Menu, Append, Browse, or Update frames. You can create other types of frames and procedures—such as Report or User frames or 4GL procedures—in the same way or with the Create operation of the Edit an Application window.

To use a frame that you create in the Edit an Application window in a Vision application, call the frame in either of the following ways:

- Write a 4GL callframe statement in escape code.

  In this case, the called frame does not appear in the application flow diagram.

- Use the Insert operation of the Application Flow Diagram Editor to insert the frame as a child of a Vision-generated frame.

Remember that only frames for which Vision generates code—Menu, Append, Browse, and Update frames—can call other frames in the application flow diagram.

In general, it is more efficient to create new frames through the Application Flow Diagram Editor, so that they are linked automatically into the application. The Edit an Application window provides a convenient method to include frames that you already have created for other ABF applications.

To create a frame or procedure through the Edit an Application window

1. With any frame as the current frame, select Catalog from the Application Flow Diagram Editor menu.

   Vision displays the Edit an Application window.

2. Select Create from the menu.

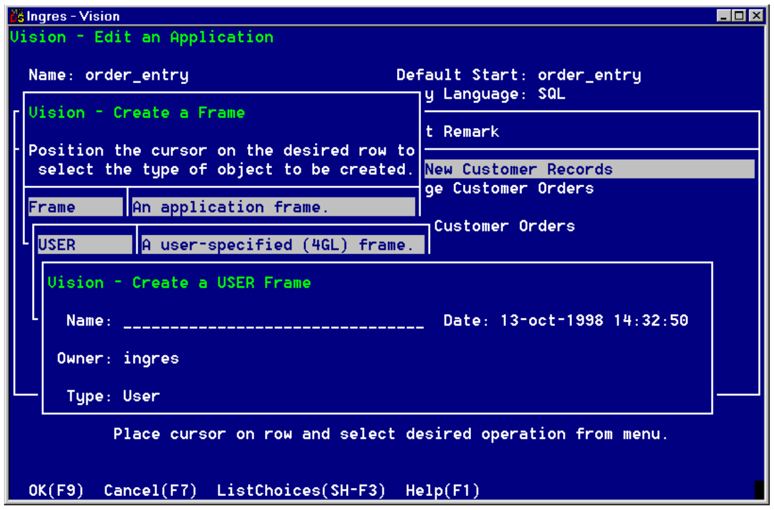
3. Select Frame or Procedure as appropriate.

4. Select one of the available frame or procedure types:

   - Frames: User, Report, QBF, or Graph
   - Procedures: 4GL, SQL (for a database procedure), or a 3GL.The list displays the names of the supported programming languages.

   Vision displays the Create a Frame or Create a Procedure pop-up for the appropriate type.

   The following figure shows an example of this window for a User frame; the window is similar for the other frame and procedure types:



```
Vision - Edit an Application

  Name: order_entry                      Default Start: order_entry
                                         y Language: SQL
  Vision - Create a Frame
                                         t Remark
  Position the cursor on the desired row to
   select the type of object to be created. New Customer Records
                                         ge Customer Orders
  Frame        An application frame.
                                          Customer Orders
   USER        A user-specified (4GL) frame.

    Vision - Create a USER Frame

      Name: ________________________________   Date: 13-oct-1998 14:32:50

     Owner: ingres

      Type: User

           Place cursor on row and select desired operation from menu.

  OK(F9)  Cancel(F7)  ListChoices(SH-F3)  Help(F1)
```

5. Enter a name for the frame or procedure and select OK.

   Vision displays the Edit a Frame Definition window or Edit a Procedure Definition window for the frame or procedure type. From this point, you define the frame or procedure as described in Defining Frames without Visual Queries (see page 159) . Begin with step 2 of the procedures for the specific frame or procedures.

Accessing Vision Functions through ABF

If you are working in the ABF environment, you can perform various Vision functions by using the methods below as an alternative to the procedures described elsewhere in this guide.

Call the Forms Editor

To call the forms editor from the Edit an Application window

1. Display the Edit an Application window for the application as described in The Edit an Application Window section.
2. Position the cursor on the name of the frame whose form you want to edit.
3. Select Edit.

   The Edit a Definition window is displayed.
4. Select FormEdit.

   The form for the frame is displayed for you to edit.

For more information about editing forms, see Defining Frames with Visual Queries (see page 117).

Create Global Components

To create or edit global components from the Edit an Application window

1. Display the Edit an Application window for the application as described in Edit an Application Window (see page 313).
2. Select Globals.

   A pop-up is displayed with a list of global types.
3. Select one of the global component types: Constants, Variables, or Record Definitions.

Define the global components as described in the Using Global Components section in Using Vision Advanced Features (see page 177).

Open Source Code Files

To edit the source code file for a User frame or procedure or a Vision-generated frame from the Edit an Application window

1. Display the Edit an Application window for the application as described in The Edit an Application Window section.
2. Position the cursor on the frame or procedure name in the Edit an Application window.
3. Select Edit.

   The Edit a Definition window appears.
4. Select Edit again.

   Vision calls the system editor and opens the source code file.

Edit the source code as described in Editing the Source Code to Create a Custom Frame (see page 239).

Accessing Vision Utilities

Use the Utilities operation on the Edit an Application window to access many of the Vision utility functions. For more information about the available utilities, see Using Vision Utilities (see page 243).

Appendix C: Template Files Reference

This section contains the following topics:

Vision uses template files to generate 4GL code for your application. This appendix lists and describes in detail the following components of the template files:

- Substitution variables
- Template language statements

For more information on using these components to modify your Vision-generated code, see Modifying Vision Code (see page 227).

Substitution Variables

Substitution variables are global variables used in template files. These variables have the form *$variable*. The Vision code generator substitutes actual values for variables contained in the template before code is generated.

Substitution variables are used in executable "template language" statements—such as the **##** generate statement—that are translated into 4GL statements in the application source code.

Substitution variables can be system defined or user defined. These two types are described in the following sections.

System-defined Substitution Variables

The code generator uses global substitution variables to represent information about the frame. You provide this information through the following Vision components:

- The application flow diagram that specifies the frames that your application uses and how those frames are called
- Visual query specifications. The visual query for each Append, Browse, and Update frame that provides basic information to generate the frame's form and database query
- Menu items to call frames
- Parameters passed between frames
- 4GL escape code; this code is incorporated into the source code file exactly as you have written it
- Frame behavior specifications that control how users interact with the frame
- Local variables that you have specified for the frame

There are two types of these substitution variables:

- *String variables*
- *Boolean variables*

The tables in the following sections list the substitution variables recognized by the code generator. You can use any of these variables when editing template files.

String Variables

String variables supply such information as the names of the Master and Detail table that the frame uses. The following variables are set to string values before the code is generated. You can use them for text substitution, or you can use them in **##**if or ##ifdef statements.

$default_return_value

Returns a default value appropriate to the frame type:

0 for integer, float or money

'' for string or date

NULL for none

$detail_table_name

Specifies the detail table in the Visual Query

$form_name

Specifies the VISUAL FORMS EDITOR form

$frame_name

Specifies the current frame

$frame_type

Specifies the type of the current frame (Browse, Update, Append, or Menu)

$locks_held

Specifies the type of locking specified in the frame behavior. Returns one of the following values:

DBMS

Optimistic

None

$master_seqfld_cname

Specifies the table column that corresponds to the sequenced field on an Append frame

$master_seqfld_fname

Specifies the form field that appears as a sequenced field on an Append frame

$master_table_name

Specifies the Master table in the Visual Query

$short_remark

Specifies the short remark entered when the frame was created

$source_file_name

Specifies the frame's source file

$tblfld_name

Specifies the name of the Master or Detail table field on the form (always "iitf")

$template_file_name

Specifies the main template file used to generate the code (for example, msappend.tf)

Boolean Variables

Boolean variables generate code based on whether or not the frame functions in a specific way; for example, whether user qualifications are allowed on an Update frame. The following variables are set to either "0" (false) or "1" (true) before the code is generated. Most of these variables are set in the Visual Query and by frame behaviors. You can use these variables in ##if or **##**ifdef statements.

$dbevent_code_exists

Specifies the frame contains On-Dbevent escape code

$default_start_frame

Specifies the default start-up frame for the application

$delete_allowed

(Update frames only) Specifies that deletes are allowed on either the Master or Detail table

$delete_cascades

Specifies that corresponding Detail table records must be deleted when a Master table record is deleted

$delete_dbmsrule

Specifies that when a user selects Delete on an Update frame with a Master and Detail table, the generated code deletes only the Master table row and allows the DBMS to delete the corresponding Detail table records

The default value is FALSE.

$delete_detail_allowed

(Update frames only) Specifies that detail table deletes are allowed

$delete_master_allowed

(Update frames only) Specifies that Master table deletes are allowed

$delete_restricted

(Update frames only) Specifies that Master table deletes are prohibited when corresponding Detail table rows exist

$insert_detail_allowed

(Update frames only) Specifies that detail table insertions are allowed

$insert_master_allowed

(Update frames only) Specifies that new Master table rows can be appended; causes an AddNew and AppendMode menu item to be generated

$join_field_displayed

Specifies that any of the join fields are displayed (used with the Update Cascades integrity rule for Update frames with Master and Detail tables)

$join_field_used

Specifies that any of the join fields are used on the form or in variables

$lookup_exists

Specifies that a Lookup table is included in the Visual Query

$master_in_tblfld

Specifies that the Master table is displayed as a table field (always false for a frame with a Detail table)

$master_seqfld_exists

Specifies that a sequenced field is specified for the Master table on an Append frame

$master_seqfld_displayed

Specifies that a sequenced field for the Master table on an Append frame is displayed on the form

$master_seqfld_used

Specifies that a sequenced field for the Master table on an Append frame is used on the form or in a variable

$nullable_master_key

(Update frames only) Specifies that any column of the Master table key is nullable

$nullable_detail_key

(Update frames only) Specifies that any column of the Detail table key is nullable

$singleton_select

Specifies that only one Master table row must be retrieved (no Next menu item is to be generated)

$tablefield_menu

Specifies that the menu items on a Menu frame are displayed as a table field

$timeout_code_exists

Specifies that the frame contains On-Timeout escape code

$update_cascades

(Update frames with Master and Detail tables) Specifies that corresponding Detail table records must be updated when a value in a join field is changed

$update_dbmsrule

Specifies that when a user selects Save on an Update frame, the generated code does not update the join fields in the Detail table, but rather allows the DBMS to apply those updates, based on changes to the corresponding columns of the Master table

$update_restricted

Specifies that the value of the join columns in a Master table record cannot be updated if corresponding Detail table records exist

$user_qualified_query

Specifies that users can enter runtime query qualifications on a Browse or Update frame

User-defined Substitution Variables

Define your own substitution variables by using the **##** define statement. The name of the substitution variable can only contain alphanumeric characters and underscores.

You can optionally specify a value when you define your own substitution variables. The value can be a string or a Boolean expression. If you do not specify a value in the ## define statement, the $*variable* is created with no value.

Vision predefines the following substitution variables in the template file "intopdef.tf". You can change the value of these substitution variables by editing the "intopdef.tf file."

$_deadlock_retry

Indicates the number of times the application retries the transaction if there is a deadlock error. The default is 2.

$_deadlock_error

Indicates the error code for a deadlock error. The default is the generic error code, 49900. If you are using dbms error codes, change the value of this variable to 4700, the dbms error code for deadlock.

$_timeout_seconds

Indicates the length of the time-out period in seconds. That is, the number of idle seconds before any on_timeout escape code is executed. The default is 300 seconds.

Template Language Statements

Template files include executable statements in a template language to tell Vision how to generate the code. The code generator executes the template language statements to produce the source code for a frame. These statements themselves do not appear in the generated 4GL code. Executable statements begin with "##."

The executable statements used by the code generator are listed in this appendix. You can add or delete statements to a template file, but you cannot change the basic format of a statement.

Formatting Rules

Template language executable statements must follow these formatting rules:

- To continue a statement on another line, end the line with a \.
- Do not include a statement terminator.
- The "##" symbol must appear in the first two columns of a line to be recognized by the code generator.
- "##" statements are not case sensitive.
- Any single or double quotes within a statement are ignored.
- There must be at least one blank between words of a statement and between the last word and a comment.
- You can include comments on an executable line by using the comment delimiter "- -". Comments on the "##" lines are for use in the template file only and do not appear in the generated code. A comment can be on a separate line or can follow an executable statement. Any code following the "- -" is ignored, so comments must appear at the end of a line. To continue comments on another line, end the line with a backslash (\) or start the next line with ## - -.

Examples—formatting rules:

```
## INCLUDE inlookup.tf -- ListChoices menuitem
## ENDIF -- $singleton_select
## -- Main template file 'iimain.tf' Processing
## -- always starts here
```

define Statement—Define a Substitution Variable

Use the define statement to define a substitution variable and, optionally, assigns a value to it. The name of the substitution variable can only contain alphanumeric characters and underscores. You can optionally specify a value, which can be a numeric expression, a character string expression, or a Boolean expression.

Character strings must be in single quotes. Character strings can be concatenated with a plus sign (+). Numeric expressions can include standard operators. An expression can be another substitution variable, in single quotes. The expression evaluates to the value of the quoted substitution variable, because the value assigned to a $*variable* is substituted everywhere, even inside quoted strings and comments.

If no expression is given for a $*variable* in the define statement, the $*variable* is created with no value. The $*variable* can be used in an ifdef statement.

This statement has the following syntax:

```
##define $variable [expression | Boolean expression]
```

Examples—define statement:

```
## define $tbl 'tf'
## define $i 1
## define $ft '$tbl' + '$i'
```

String concatenation: this evaluates to tf1 if the first two are defined as shown:

```
## define $ft '$tbl$i'
```

Same as above:

```
## define $i $i+1
## define $i $a+$b+\
$c+$d
```

generate Statement—Generate 4GL Source Code

Generates 4GL source code based on specifications in the frame definition or escape code.

This statement has the following syntax:

```
## generate code_type {argument,...}
```

The *code_type* can be any of the following types:

check_change

Generates 4GL inquire_forms statements based on the setting of the Update Integrity rule for Update frames. The only argument this statement accepts is join_fields. This statement generates a 4GL inquire_forms statement and other 4GL code to check whether the value of any of the join columns has been changed for the Master table of an Update frame.

Syntax:

```
##generate check_change join_fields
```

copy_hidden_to_visible join_fields

Undoes changes to the value of the Master and Detail table join columns. Used in Update frames when the Update Integrity Rule restricts updates on the join column.

Syntax:

```
##generate copy_hidden_to_visible join_fields
```

help

Generates a 4GL help_forms statement, copies the named help file from the template file directory into the application's source directory, and gives the help file the same name as the ".osq" source file.

Vision creates one or more help files for each frame based on the frame type and frame definition. You can edit the help files that Vision creates, as discussed in the Vision-Generated Help Files section in Modifying Vision Code (see page 227).

Syntax:

```
##generate help helpfilename
```

For example, the following statement moves the named help file to the source directory and gives it the same name as the frame:

```
##generate help fgmdupda.hlp
```

hidden_fields

Defines local variables, or hidden fields, for the frame. The name of each hidden field that Vision generates begins with "iih_." Local variables can be any of the following:

- hidden fields (columns which are marked "n" on the visual query)
- column names for primary keys and join fields in Update frames
- column names for non-displayed sequenced fields in Append frames
- columns which are marked "v" on the visual query
- user-defined local variables
- user-defined local procedures

Syntax:

```
##generate hidden_fields
```

load_menuitems

If a menu frame is in table field style, then this statement populates the table field.

Syntax:

```
##generate load_menuitems
```

local_procedures

Generates user's local procedure code. The optional declare keyword generates the declare section for the local procedure.

Syntax:

```
##generate local_procedures [declare]
```

lookup

Generates most of the code for the ListChoices menu item. This code tests for whether the cursor is on a Lookup table activation field. If so, then a 4GL callframe look_up ( ) statement is issued.

Syntax:

```
##generate lookup
```

query

Generates the query appropriate for the frame type and visual query specifications. The full syntax of this statement is:

```
## generate query query_type VQtable [repeated] [noterm]
```

query_type

Is any of the following:

- select for Browse and Update frames
- insert for Append frames (and Update frames that allow inserts of new Master or Detail table records)
- update for Update frames
- delete for Update frames that allow deletions of Master or Detail table records

VQtable

Is one of the following:

- Master table only
- Detail table only
- (Select queries only) Master_Detail for generating a Master/Detail select query

repeated

Is an optional keyword that generates repeat queries (repeat queries run faster than non-repeat queries on second and subsequent runs).

All Vision-generated queries use this keyword, except those that use a 4GL qualification function.

noterm

Is an optional keyword that causes the query statement to be generated without a semi-colon (;) as a statement terminator

Queries that have an attached submenu use this keyword. For example:

```
## generate query insert master
```

tells the code generator to formulate a 4GL insert statement to append data to the Master table. This statement appears in the Append frame's template file.

Another example:

```
## generate query update master repeated
```

generates a repeat query to update the Master table.

set_default_values simple_fields

Assigns default values for displayed fields of the Master table on an Append frame, based on values specified in the Assignment/Default field of the Visual Query.

Syntax:

```
##generate set_default_values simple_fields
```

set_null_key_flags master|detail

Handles update and delete statements for Master or Detail tables that have nullable keys. Used with an ##IF $nullable_master_keys or ## IF $nullable_detail_keys statement.

Syntax:

```
##generate set_null_key_flags master | detail
```

user_escape

Generates a begin-end block to insert 4GL escape code at the appropriate location in the source code file. The escape code is inserted exactly as written. The code generator does not scan it for any template language statements, substitution variables, or logicals or environment variables.

The full syntax of this statement is:

```
## generate user_escape escape_type
```

In general, the value of *escape_type* corresponds to the type of the escape code as displayed in Vision except the *escape_type* name contains underscores (for example, "form_start") instead of dashes ("Form-Start"). The following cases are exceptions:

- The value "after_field_activates" represents both the After-Field-Change and After-Field-Exit escape types.
- The value "before_field_activates" represents the Before-Field-Entry escape type.

user_menuitems

Generates a menu activation block for each menu item that you have specified for a frame (that is, each of the frame's child frames in the Application Flow Diagram). The block includes the necessary callframe statements and any Menu-start or Menu-end escape code you have specified.

if then else endif Statement—Process Conditional Statements

Processes a set of statements based on the value of a Boolean substitution variable or an expression that evaluates to a Boolean value:

- Processes the statements if the variable returns a value of TRUE (TRUE is equal to the string "1")
- If the variable returns a value of FALSE, processes the statements in an else clause, if such a clause is included (FALSE is equal to the string "0")
- If the variable returns a value of FALSE and no else clause is included, ignores all statements up to the endif statement
- If a variable name ($*variable*) is used in a Boolean expression to compare with a string value, the variable must be in single quotes ('$*variable*').
- The if-endif statements are required; the else clause is optional. If-endif blocks can be nested to 20 levels deep.

This statement has the following syntax:

```
## if $variable | Boolean expr then {statement}
[## else {statement}]
## endif
```

Examples—if then else endif statement:

- The following statement evaluates to True if the substitution variable $*frame_name* is the string 'myframe':

```
## IF ('$frame_name'='myframe') THEN
   code to execute for 'myframe'
## ENDIF
```

- The following statements generate a Lookup table if the variable $lookup_exists is defined:

```
## IF $lookup_exists THEN
## GENERATE LOOKUP
## ENDIF
```

- The following statements generate code according to whether the frame is a big frame or not.

```
## if (('$frame_name'='big'+'one') or ($bigframe=TRUE))
## THEN
   code to execute for big frame
## ELSE
   code to execute for not big frame
## ENDIF
```

ifdef Statement—Process a Set of Statements with Variables or Logicals

Processes a set of statements depending on whether a $*variable* or $$*logical* is defined. The statements within the ifdef-endif block are executed only if the $$*logical* or $*variable* has been defined. The statements are executed even if the $*variable* is created with no value.

The statements in the optional else clause are executed if the $$*logical* or $*variable* is not defined.

This statement has the following syntax:

```
## ifdef $variable | $$logical {statement}
[## else {statement}]
## endif
```

Example—ifdef statement:

The following statements generate a 4GL comment stating the value of II_TFDIR, if it is defined:

```
## IFDEF $$II_TFDIR
/* II_TFDIR: $$II_TFDIR */
## ENDIF
```

ifndef Statement— Process a Set of Statements without Variables or Logicals

Processes a set of statements if a $*variable* or $$*logical* is not defined.

The statements within the ifndef-endif block are executed only if $$*logical* or $*variable* has not been defined. Statements in the optional else block are executed if the $$*logical* or $*variable* has been defined. A $*variable* can be defined without a value. In this case, the statements in the ifndef block are not executed, because the $*variable* is defined.

This statement has the following syntax:

```
##ifndef $$logical | $variable {statement}
[## else {statement}]
## endif
```

Example—ifndef statement:

```
##ifndef $mode
/* Caution: '$mode' has not been defined */
##else
/* The value of '$mode' is $mode */
##endif
```

include Statement—Include a Template File

Includes another template file. You can nest include statements up to 20 levels deep.

The argument *filename* can be any of the template files beginning with "in." The *filename* can be unquoted, or enclosed in single or double quotes. The *filename* cannot include a directory path. The code generator searches for the included file in the directory specified by the II_TFDIR logical or environment variable. If you have not specified II_TFDIR, or if the file is not found in the specified directory, Vision searches for the file in the default location. For more information about specifying template file locations, see Template File Locations (see page 229).

This statement has the following syntax:

```
## include filename
```

Examples—include statement:

The following statement *includes* the template file "inend.tf" into a frame template:

```
## include inend.tf
```

The following statement tells the code generator to include the template file to generate a Next menu item:

```
## include innxmn.tf
```

undef Statement—Undefine a Substitution Variable

Use the ##undef statement to undefine a substitution variable defined with the ##define statement. You can only undefine a user-defined substitution variable.

This statement has the following syntax:

```
##undef $variable
```

Example—undef statement:

The following statements indicate the value of the variable $status, and undefine it, if necessary.

```
##ifdef $status
/* '$status' is defined as $status */
##undef $status
/* '$status' is now undefined */
##else
/* '$status' is undefined */
##endif
```

Appendix D: Vision Architecture

This section contains the following topics:

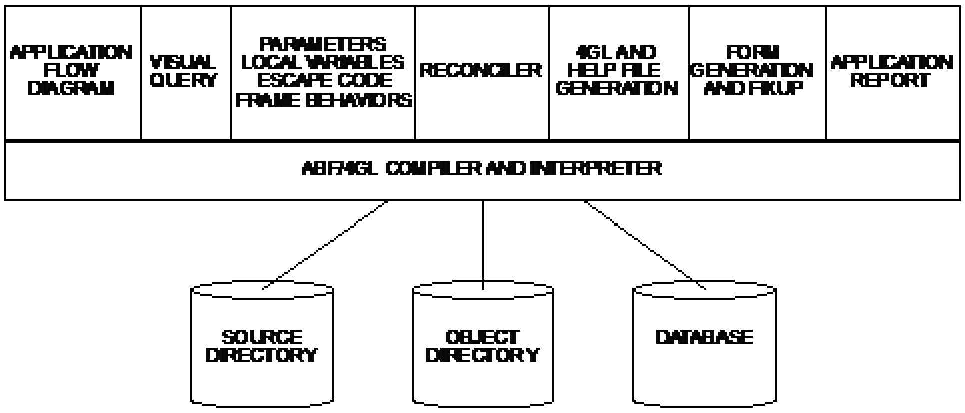
This appendix describes how:

- Various components of Vision are related to each other and to ABF
- Vision 4GL code generator uses those components to generate the source code for your Vision applications
- Vision-generated code handles deadlock errors
- Vision Reconcile utility incorporates changes in table definitions

Vision Components

Vision uses the compiler and interpreter already in place as part of ABF. In this way, Vision functions in an Ingres environment as an extension of ABF, allowing you to combine the functions of the two tools, as described in Accessing Vision through ABF (see page 313).

The following figure shows how the various Vision and Ingres components interact:


APPLICATION FLOW DIAGRAM
VISUAL QUERY
PARAMETERS LOCAL VARIABLES ESCAPE CODE FRAME BEHAVIORS
RECONCILER
4GL AND HELP FILE GENERATION
FORM GENERATION AND FIXUP
APPLICATION REPORT
ABF/4GL COMPILER AND INTERPRETER
SOURCE DIRECTORY
OBJECT DIRECTORY
DATABASE

Each of Vision's components is used to create the 4GL source code and forms for an application. These components include:

- The application flow diagram that specifies the frames that your application uses and how those frames are called
- The visual query for each Append, Browse, and Update frame that provides basic information to generate the frame's form and database query
- Additional information—parameters, local variables, 4GL escape code and frame behavior specification—used to generate queries and forms or specify how the user interacts with a frame

  Any 4GL escape code is included in the generated source code exactly as you have written it.
- The Reconcile utility to modify visual queries whose database tables have been changed (by adding a new column, for example)

  See How the Vision Reconcile Utility Works (see page 339) for more information about the reconciliation process.
- The Vision code generator that creates:
  - A source code file for each frame

    The code generator creates this file by combining a generic template for a frame type with information about a specific frame of that type. This information is provided by the visual query and other components discussed above.
  - A copy of a standard user help file for each frame, based on the frame's type

    See How Vision Generates Code (see page 337) for more information about how Vision generates source code and help files.
- The Vision form generation facility that creates a form for each generated frame

  Vision determines the fields for the form based on the visual query and application flow diagram. Vision modifies the form when these specifications change.
- The application report that provides a summary of information about each frame in an application

The output from these components is processed through the ABF/4GL compiler and interpreter, as with any ABF/4GL application. After the application is compiled:

- Create an image file that contains the compiled code; this file is written to the object directory.

  Ingres makes no distinction between the code that Vision generates and any other 4GL code that your application uses. You can edit Vision-generated source code as described in Modifying Vision Code (see page 227).

- The frames and forms that you created through Vision are stored as database objects in the appropriate catalogs.

  Ingres treats these objects the same as any other database objects, such as reports or join definitions, that you have used with frames for which Vision did not generate the 4GL code.

How Vision Generates Code

Vision creates the source code, help files, and forms for the Append, Browse, Menu, and Update frames in your application.

Vision combines your visual queries and other frame definition specifications with a set of frame templates and standard help files.

Vision generates 4GL source code and forms when you create a frame, and regenerates the code and forms as necessary as you build your application. For details on when code is regenerated, see Regeneration of Source Code and Forms (see page 260).

The Vision code generator creates:

- A source code file for each frame

  The code generator creates this file by combining a generic template for a frame type with information about a specific frame of that type. You provide this information through the visual query and other frame definition specifications.

- A copy of a standard user help file for each frame, based on the frame's type

- A form for each generated frame.

  Vision determines the fields for the form based on the visual query and application flow diagram. Vision modifies the form when these specifications change.

The output from these components is processed through the 4GL compiler and interpreter, as with any 4GL application.

The generated 4GL code is contained in an ".osq" file that generally has the same name as the frame. This file is stored in the application source directory.

VMS: After the code has been generated successfully, Vision purges all but the three latest copies of the ".osq" file.

After the application is compiled, the frames and forms that you created through Vision are stored as database objects in the appropriate catalogs.

Ingres treats these objects the same as any other database objects, such as reports or join definitions, that you have used with frames for which Vision did not generate the 4GL code.

How Vision Handles Deadlocks

Deadlock occurs when two transactions are holding locks that each block the other transaction from completing. When deadlock occurs, Ingres aborts one of the transactions, so that the other transaction can complete. The user whose transaction was aborted receives an Ingres error.

In a Vision application, deadlock can occur when the Locks Held on Displayed Data frame behavior is set to DBMS or shared locking for an Update frame. For example, deadlock occurs when multiple users hold a shared lock on the same data and each selects Save.

The Vision-generated code tests Ingres errors from the DBMS to see if deadlock has occurred. When Vision detects a deadlock error, it retries the transaction twice. If the transaction still cannot complete after three tries, the transaction is aborted.

By default, Vision tests for the generic error number, 49900. If your application uses DBMS error numbers, you must change the code to test for the DBMS error number, 4700. Change the value of the substitution variable $_deadlock_error, which is located in the intopdef.tf template file. You can also change the number of retries by setting the substitution variable $_deadlock_retry, also located in the intopdef.tf template file.

Vision does not hide the DBMS error message. The Vision application user sees the Ingres error message for deadlock. To hide Ingres errors from your application users, you must write your own error handler, and call it with the function IIseterr(). See the 4GL part of this guide for details.

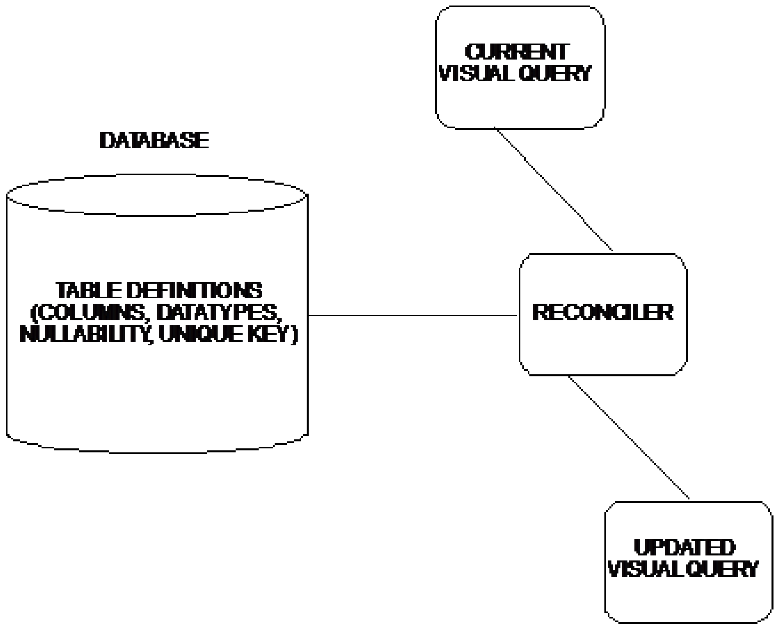
See the *Database Administrator Guide* for more information about locking in Ingres.

How the Vision Reconcile Utility Works

If you make changes to the definitions of the tables in your database—by adding a column, for example—the visual queries for the Vision frames that use those tables can no longer reflect the tables accurately. Therefore Vision provides a Reconcile utility that:

- Detects inconsistencies between visual queries and the tables that they use
- Lets you tell Vision how to resolve those discrepancies

The following figure shows the reconciliation process:


CURRENT VISUAL QUERY
DATABASE
TABLE DEFINITIONS (COLUMNS, DATATYPES, NULLABILITY, UNIQUE KEY)
RECONCILER
UPDATED VISUAL QUERY

The Vision Reconcile utility functions as follows:

1. When you select Reconcile, Vision compares the current visual queries for the generated frames of an application against the definitions stored in the database for the tables used by those frames.
2. Vision informs you of any discrepancies based on:
    - Columns that have been added or deleted
    - A column's data type that has been changed
    - A column that has been made nullable or not nullable
    - A table whose unique key has been changed
    - Entire tables that have been deleted from the database

3. Vision gives you the opportunity to see how the reconciliation process can affect your visual query.
4. Based on your instructions, Vision changes the visual query to match the new table definition.

For a description of the changes that Vision can make to a visual query and procedures for using the Reconcile utility, see Reconciling Tables and Frame Definitions (see page 251).

PART 3: Applications-By-Forms

Chapter 12: Overview of ABF

This section contains the following topics:

ABF is a workshop for creating customized forms-based applications. It takes you through the development of a new application by presenting a series of forms that you fill in and menus that have operations you can choose. The resulting application uses standard Ingres forms and menus to access a database and perform a variety of operations, including queries, updates, and reports.

ABF allows you to define, test, and run fully developed applications without having to use a conventional programming language. To enhance the applications with specialized processing, you can write scripts using 4GL. For a summary of the relationship between ABF and 4GL, see Overview of Tools and Languages (see page 35). For detailed information on 4GL, see the 4GL part in this guide.

Benefits

With ABF, you can create applications quickly out of readily available building blocks. Linking together queries, forms, and reports supplied by ABF provides you with:

- Access to Ingres tools

  ABF provides easy access to the Ingres tools you need to create an application, such as the VIFRED and the system editor.

  The FormEdit operation places VIFRED at your disposal for creating a form that can accept user input or display data from the database. When you finish, VIFRED returns you to ABF.

  The Edit operation provides access to your system editor to code 4GL source files. When you create 4GL specifications for user-specified frames, ABF automatically compiles any newly created or revised 4GL files before running your application. ABF stores the source file in the directory you designate, and it incorporates the compiled version into the system catalogs.

  These operations are described in more detail in Building Applications (see page 351), and ABF Development Example (see page 453).

  ABF also allows you to incorporate Ingres tools such as QBF and RBF into your applications, and make Ingres tools available to the application user.

- A code manager for most files related to an application

  You can use ABF without worrying about the location and management of source files, object files, linkage programs, compilers, editors, and the other tools of conventional programming. ABF keeps track of the names and locations of all the files related to your completed application.

- A dynamic test environment

  You can debug and test the application during development. ABF allows you to test the application in small, manageable pieces—a much easier process than testing the entire application at one time.

- Default actions

  ABF provides default actions if the application tries to access the values in an undefined object. For example, if the application tries to access a frame that is undefined, ABF provides a message that says that the specified frame is undefined, and gives the user several options on how to continue.

Types of Frames

ABF has three types of frames:

User-specified

Custom frames created by the application developer through ABF and 4GL

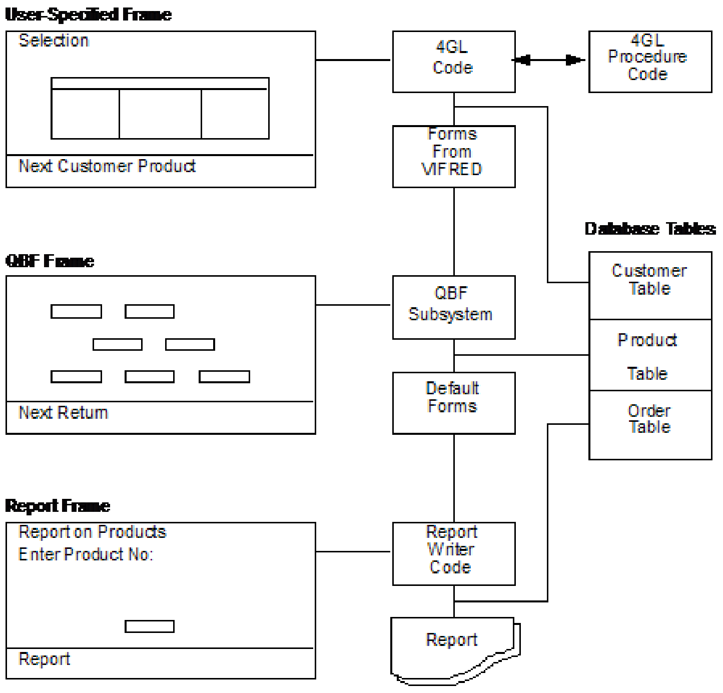
QBF

Frames that perform database queries. ABF calls QBF to create these frames.

Report

Frames that display or print reports. ABF calls Report-Writer and RBF to create these frames.

All frames, regardless of type, are made up of the same basic components and all access data in the database tables. The following figure shows a sample of each type of frame, with its connection to the Ingres tool that defines it:



The following sections illustrate and discuss each frame type.

User-specified Frames

A *user-specified frame* is defined by the application developer. After you create the user-specified frame in ABF, there are three steps in defining the frame:

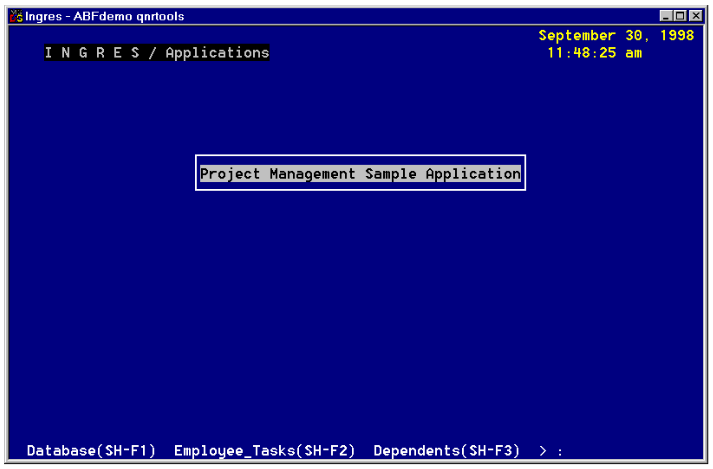
- Specify the name and operations to be performed for each item on the menu of that frame with 4GL.
- Design the form for the frame with the FormEdit operation. (See Forms below.)
- Create a 4GL specification defining each menu operation along with statements that determine what happens when the user chooses each operation.

These steps can be performed in any order.

When running, the frame displays the form and menu in a window. When the user chooses a menu operation, the 4GL code for that operation is executed, causing the application to perform the associated actions, including:

- Run other frames and procedures, and start up Ingres tools.
- Run external applications or system programs, display your own help files, and perform specific data manipulations.

The following figure shows the main menu frame of the sample ABF application described in The ABF Demo Program (see page 515). This is a user-specified frame that consists of a menu of operations and a form containing several lines of trim.



```
                                                    September 30, 1998
  I N G R E S / Applications                          11:48:25 am

                    Project Management Sample Application

 Database(SH-F1)  Employee_Tasks(SH-F2)  Dependents(SH-F3)  > :
```

For more information on user-specified frames, see Building Applications (see page 351), and ABF Development Example (see page 453).

QBF Frames

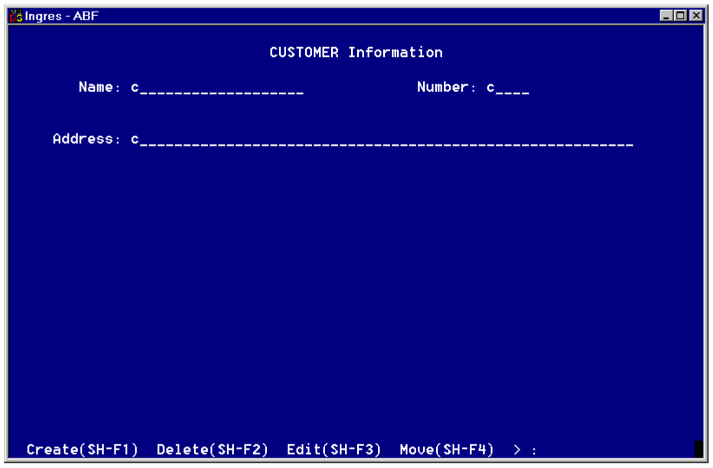
Whenever you include a *QBF frame* in an application, you are specifying that the application use QBF to access the database for queries. Use QBF (QBF) frames for operations that directly access the database, such as adding new rows to a table or retrieving data from a series of tables. When defining a QBF frame, specify:

- The table or JoinDef (join definition)
- The form with which to run QBF
- The command line flags to be used in the call to QBF

When the frame is activated, QBF begins executing as specified by the associated form, table or JoinDef, and flags.

You can use a QBF default form as is, or you can enhance it by using the FormEdit operation to call VIFRED. (See Forms (see page 348).)

The following figure shows an example of a QBF frame, with default QBF menu operations. QBF also provides its own standard field and key activations. You do not need to use 4GL code to specify the operations of a QBF frame.



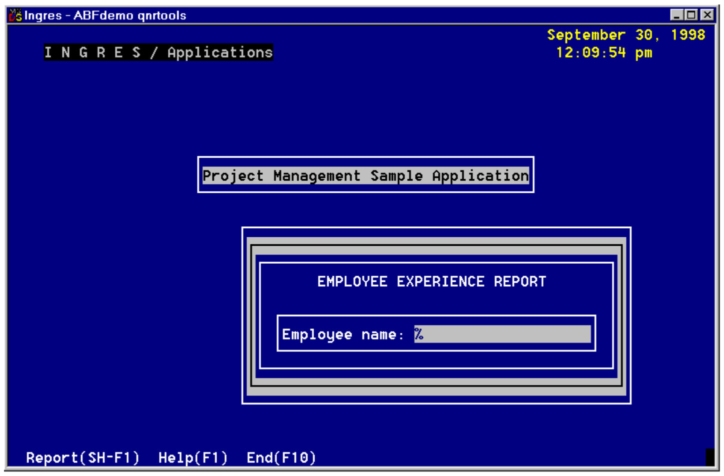
```
Ingres - ABF

                              CUSTOMER Information

       Name: c___________________              Number: c____

    Address: c_________________________________________________________

  Create(SH-F1)  Delete(SH-F2)  Edit(SH-F3)  Move(SH-F4)  > :
```

For a complete description of QBF, see *Character-based Querying and Reporting Tools User Guide.* For more information on using QBF frames with ABF, see Building Applications (see page 351), and ABF Development Example (see page 453).

Reports and Report Frames

A report object in ABF is a report specified with RBF or Report-Writer commands. You can use an existing report or the system editor to create a new report specification file as you fill out the report definition in ABF.

A *report frame* consists of a report and a menu for running it. The frame can include a form on which the user can enter one or more values used by the report at run time. You can create the form using the FormEdit operation to call VIFRED. In the following example, the user enters the employee name to request a report for that employee. The percent sign (%) is used to indicate all employees.



```
I N G R E S / Applications                    September 30, 1998
                                               12:09:54 pm

           Project Management Sample Application

                 EMPLOYEE EXPERIENCE REPORT

                 Employee name: %

Report(SH-F1)  Help(F1)  End(F10)
```

The report frame has a set of default menu operations associated with it. No 4GL code is necessary for a report frame.

For more information on report frames, see Building Applications (see page 351), and ABF Development Example (see page 453).

Forms

You create forms using the FormEdit operation. This lets you design and change the layout of a form, define validation criteria (edit checks) and error messages, and specify the way the form is displayed in the window.

The FormEdit operation is described in more detail in Building Applications (see page 351), and ABF Development Example (see page 453).

Directories for Source Code and Application Components

ABF stores the code for your applications in directories on the disk. Source code files and the compiled versions of the application's components are placed in separate directories.

ABF never deletes or purges files from the application's source code directory. It is your responsibility to remove any unwanted source files from this directory.

ABF does purge and clean up unneeded files in the object code directory.

Establishing a Directory for Source Code

The Source Directory field on the Create an Application frame (described in How to Create the Application (see page 52)) contains the full directory path name for the directory where the source code files for the application resides. This includes the 4GL code for user-specified frames, source code for procedures, and the report specification files.

The initial value in this field is the name of the directory from which you started ABF. You can change this to any directory for a given application.

After you create an application component, avoid changing the directory specification. If you decide to change the directory name:

- Make sure the new directory actually exists.
- Create the directory at the operating system level, if necessary, before running ABF.
- Move any source code files created in the original directory to the new source code directory.

To modify directory specifications, use the Application Defaults frame, described in Building Applications (see page 351).

Establishing a Directory for Compiled Application Components

Before you start ABF, a parent directory must be established on the system for application subdirectories. Typically, the system administrator sets up the parent directory by defining ING_ABFDIR. For further information, see ABF Architecture (see page 501).

Once the parent directory is created, ABF establishes an object code subdirectory to store the compiled versions of each application's components. This subdirectory is established when you choose Create from the Applications Catalog menu (described in How to Create the Application (see page 52)). This guide refers to this subdirectory as the application's object code directory.

Chapter 13: Building Applications

This section contains the following topics:

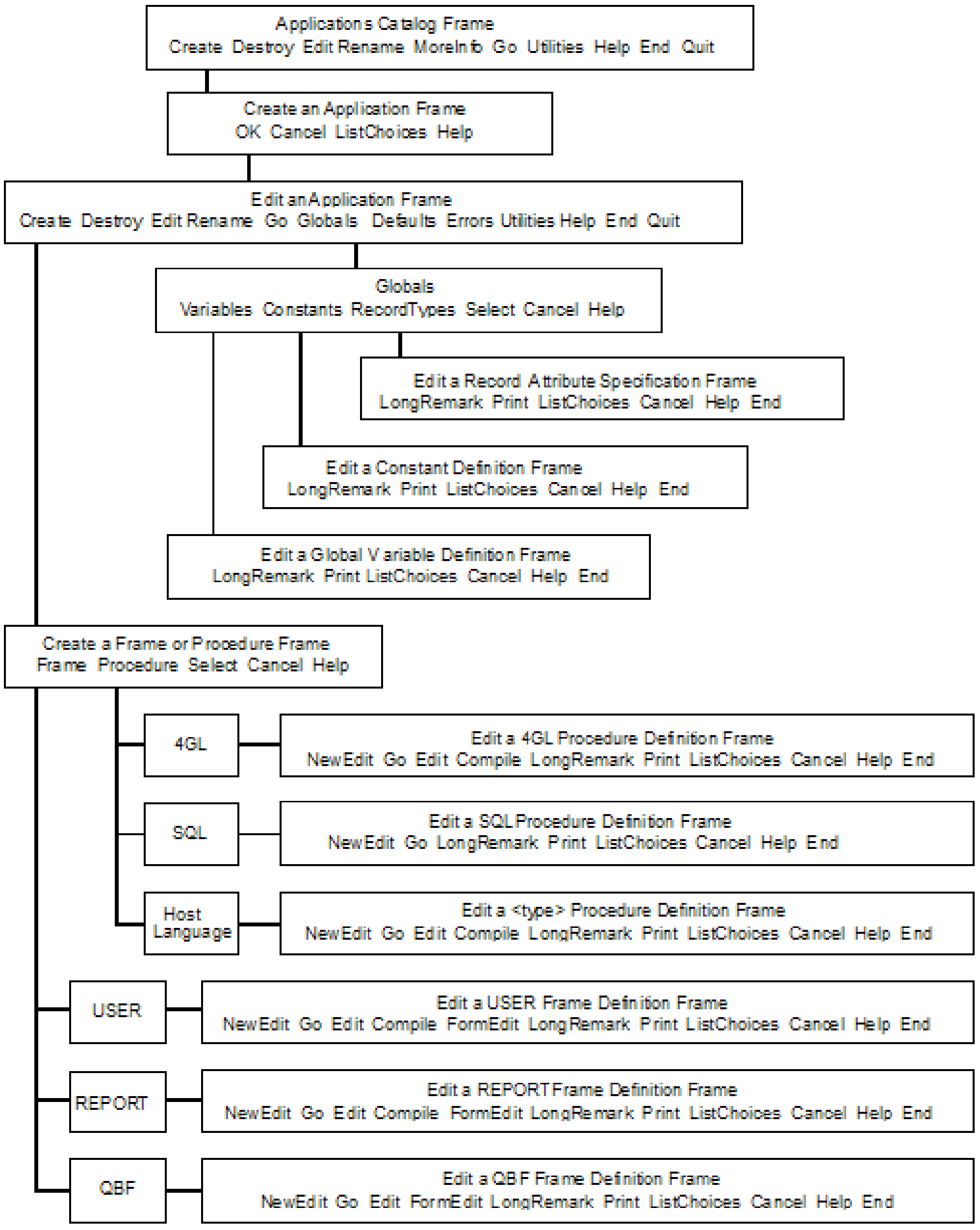
This chapter takes you through the complete development of an ABF application, from starting ABF through running the finished program. It describes all the components of an ABF application and tells you how to create, examine, and modify them. It presents ABF frames in the order in which you use them as you develop an application, and covers the purpose, fields, and menu options of each frame, as well as directions for using it. (The Applications Catalog and Create an Application frames are described in Overview of Tools and Languages (see page 35).)

As you go through this chapter, see the sample instructions in ABF Development Example (see page 453). Together they help you plan your own applications.

See the 4GL part of this guide for more information about using 4GL to enhance all the components of your application.

The ABF Development Process

This figure shows the flow of the ABF development process:


Applications Catalog Frame
Create Destroy Edit Rename MoreInfo Go Utilities Help End Quit
Create an Application Frame
OK Cancel ListChoices Help
Edit an Application Frame
Create Destroy Edit Rename Go Globals Defaults Errors Utilities Help End Quit
Globals
Variables Constants RecordTypes Select Cancel Help
Edit a Record Attribute Specification Frame
LongRemark Print ListChoices Cancel Help End
Edit a Constant Definition Frame
LongRemark Print ListChoices Cancel Help End
Edit a Global Variable Definition Frame
LongRemark Print ListChoices Cancel Help End
Create a Frame or Procedure Frame
Frame Procedure Select Cancel Help
4GL
Edit a 4GL Procedure Definition Frame
NewEdit Go Edit Compile LongRemark Print ListChoices Cancel Help End
SQL
Edit a SQL Procedure Definition Frame
NewEdit Go LongRemark Print ListChoices Cancel Help End
Host Language
Edit a <type> Procedure Definition Frame
NewEdit Go Edit Compile LongRemark Print ListChoices Cancel Help End
USER
Edit a USER Frame Definition Frame
NewEdit Go Edit Compile FormEdit LongRemark Print ListChoices Cancel Help End
REPORT
Edit a REPORT Frame Definition Frame
NewEdit Go Edit Compile FormEdit LongRemark Print ListChoices Cancel Help End
QBF
Edit a QBF Frame Definition Frame
NewEdit Go Edit FormEdit LongRemark Print ListChoices Cancel Help End

Starting and Using ABF

You can start ABF from the Ingres Menu or by using the abf command at the operating system prompt. Each method is described in the following sections.

Start ABF from the Ingres Menu

You can start ABF from Ingres Menu *if your installation does not include Vision.* If your installation includes Vision, you cannot start ABF directly from Ingres Menu, but you can edit your ABF applications from Vision. For details, see Accessing Vision through ABF (see page 313).

To start Applications-By-Forms from the Ingres Menu

1. Call the Ingres Menu. At the system prompt, enter:

   ```
   ingmenu [nodename::]dbname
   ```

   where *dbname* is the name of the database you are using. If you are using a remote host, *nodename* is the name of the remote node.

2. At the Ingres Menu, highlight Applications and choose Select.

   - If your installation does not include Vision, the ABF Applications Catalog frame is displayed. You are now in ABF.
   - If your installation includes Vision, the Vision Applications Catalog frame is displayed. You are now in Vision. To edit your ABF applications from Vision, see Accessing Vision through ABF (see page 313).

Start ABF Using the ABF Command

You can start ABF from the operating system prompt.

To start ABF from the operating system prompt

1. At the operating system prompt, type:

   ```
   abf [nodename::]dbname [applicationname [framename]]
   ```

   where *dbname* is the name of the database you are using. If you are using a remote host, *nodename* is the name of the remote node.

   You can optionally specify the name of the application you are working on, as well as the name of a specific frame.

   You can also specify other command-line parameters to access ABF in various ways. For the full syntax of the abf command, see Using Vision or ABF in a Multi-Developer Environment (see page 1381).

2. Press Return.

   If an application name was specified, you see the Edit an Application frame for that application. If not specified, you see the Applications Catalog frame, where you can select an application. (The Applications Catalog window is discussed in Overview of Tools and Languages (see page 35).)

Using ABF Fields and Menus

Follow these directions to navigate in the fields and use the menu on any ABF frame. For information on key mappings, see Help on ABF Frames (see page 355).

To use table fields:

- Use the keys mapped to the FRS commands Upline and Downline to highlight the row you want.
- Use the keys mapped to Find, Top, and Bottom to move around in a table field:
  - Use Find to locate a specified string.
  - Use Top to scroll to the top of the table field.
  - Use Bottom to scroll to the bottom of the table field.
- Press an alphanumeric key to find the next row that begins with that character. For example, press q to find the next row that starts with the letter "q."

To use simple fields:

- Use the key mapped to NextField (usually the Tab key) to move from field to field.
- Use the key mapped to PreviousField (usually Shift-Tab for PCs, or CTRL-P for UNIX and VMS) to move backward to the previous field.

To use the menu:

1. Press the Menu key (usually PF1 for UNIX or VMS, or Esc for PCs) to move the cursor to the menu line at the bottom of the frame. To find out how this key is mapped in your system, see the *Character-based Querying and* Reporting *Tools User Guide.*
2. When a frame has more menu items than it can display at one time, a right-angle bracket (<) appears at the right of the menu line. Press the Menu key again to toggle between the parts of the menu.
3. To choose the menu operation from the menu line, type the first one or two unique letters and press Return, or press the associated function key.

To leave the menu line without making a choice, press Return.

Help on ABF Frames

ABF provides context-sensitive help. This means that you can obtain help on your current task or current field. Select the Help menu operation from the menu line or press the Help key to enter the Help Utility, a feature of all Ingres tools.

A text file gives the purpose of the frame, field descriptions, key mappings, and definitions of the frame's menu operations. To display the parts of the Help window, use the keys mapped to the FRS commands Upline and Downline to scroll through the file.

The Help Utility menu offers the following operations:

Keys

Describes the current mapping of function, control, and arrow keys

Field

Displays a list of valid values for a field or display format, data type, and validation check, if any, for a field

SubTopics

On some Help menus, offers additional information related to the current Help topic

NextPage

Displays next page of the Help window (same as ScrollUp function key)

PrevPage

Displays previous page of the Help window (same as ScrollDown function key)

Help

Displays the type of help available

End

Exits from any Help window to the previous frame

Key Mappings

Select Keys from the Help menu to see a list of the current key mappings for the frame for which you called Help. The Command Key Mappings for Help frame displays default mappings; you can change these during the development process for ABF or for any applications you develop. For further information on key mapping, see:

- *Character-based Querying and Reporting Tools User Guide*
- The set_forms statement in the 4GL part of this guide

For further information about the Help Utility and its menu operations, see the *Character-based Querying and Reporting Tools User Guide*.

Exit ABF

To leave ABF, use one of these methods:

- Select Quit from the menu of a frame when it is available. You are prompted to save your changes, then exit to the operating system prompt.
- Select End from the menu of any frame. You return to the previous frame. Continue to select End to back out of each frame until you get to the Applications Catalog frame. Select End or Quit to exit to the operating system prompt.

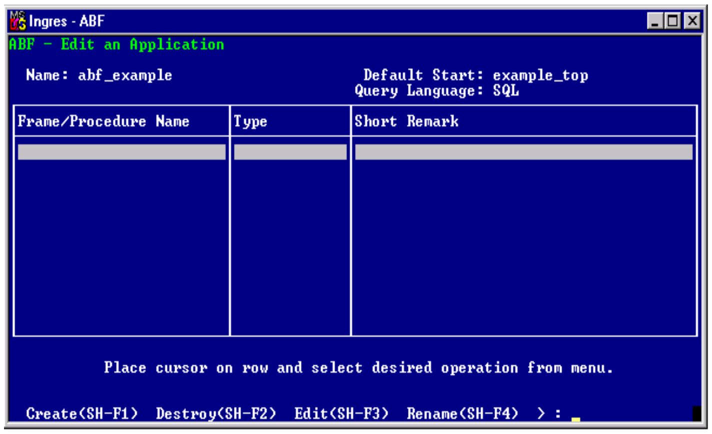
Creating an Application

To create an application, follow these basic steps:

1. Plan the application. This is described in Overview of Tools and Languages (see page 35).
2. Start ABF as described in the Starting and Using ABF section. This displays the Applications Catalog frame, described in Overview of Tools and Languages (see page 35).
3. Choose Create from the Applications Catalog frame. This displays the Create an Application frame, described in Overview of Tools and Languages (see page 35).
4. Choose OK. This displays the Edit an Application Frame described in the following section.

The Edit an Application Frame

When you create or select an application for editing, ABF displays the Edit an Application frame:



```
Ingres - ABF
ABF - Edit an Application

  Name: abf_example                              Default Start: example_top
                                                Query Language: SQL

 Frame/Procedure Name      Type            Short Remark

            Place cursor on row and select desired operation from menu.

  Create(SH-F1)  Destroy(SH-F2)  Edit(SH-F3)  Rename(SH-F4)  > :
```

The Edit an Application frame is a catalog of frames and procedures defined for the application. The columns list the frame and procedure names, frame types, and a short remark. If this is a new application and you have not yet created frames and procedures, the list is empty, as in the figure. You can use the FRS keys mapped to Upline and Downline to scroll through this list.

To create a new frame or procedure, select the Create operation. To use the other menu operations on any component in the list, highlight the row in which the component appears, then select the menu operation.

To create or edit record types, global variables, and constants, use the Globals menu operation. These options are discussed in more detail in The Globals Submenu of the Edit an Application Frame (see page 394).

The Edit an Application Menu

The following table shows the operations available from the Edit an Application frame. The selections are similar to those for the Applications Catalog. Create, Destroy, Edit, Rename, Go, and Globals are discussed in more detail below.

Create

Begins creation of a new frame or procedure. ABF displays a series of frames that allow you to specify the name and type of the component.

Destroy

Destroys the selected frame or procedure definition and remove it from the database. Prompts you to confirm the decision.

Edit

Edits the selected frame or procedure. Edit displays the Edit a Definition frame for the selected component.

Rename

Renames the selected frame or procedure. ABF prompts you for a new name. Press Return to cancel the Rename request.

Go

Runs the application for testing, whether complete or not, or before you create the executable image

Globals

Brings up the Globals menu to create or edit application global variables, constants, or record types

Defaults

Displays application-specific defaults, including Source Code directory. Alternate entry to the Application Defaults frame.

Errors

Displays uncorrected compilation errors for the application

Utilities

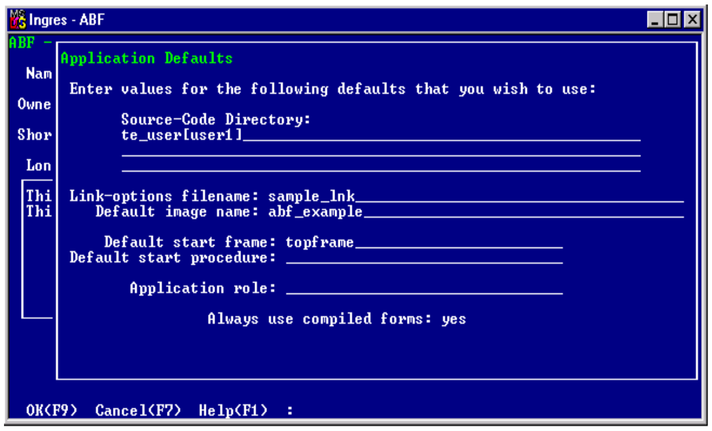
Brings up the Utilities menu

Help, End, Quit

Perform standard operations

Application Defaults Frame

Use the Application Defaults frame to establish default values for application parameters that control editing, compilation and start-up for frames:



```
ABF -
      Application Defaults
  Nam
      Enter values for the following defaults that you wish to use:
 Owne
            Source-Code Directory:
 Shor       te_user[user1]___________________________________________
            _________________________________________________________
  Lon       _________________________________________________________

  Thi Link-options filename: sample_lnk______________________________________
  Thi    Default image name: abf_example_____________________________________

          Default start frame: topframe_________________________
      Default start procedure: _________________________________

             Application role: _________________________________

                      Always use compiled forms: yes


  OK(F9)  Cancel(F7)  Help(F1)  :
```

This frame has the following fields, which appear with default values.

Source-Code Directory

(Required) Specifies the full directory path for the directory to contain any application source files. Make sure the directory exists before you enter its name here. This is a scrollable field.

Link-options Filename

Specifies the name of the file to contain additional link commands required by the application (if any). The default location for the file is the current directory, or you can specify a path name for a file in another directory. This overrides the value specified by the ING_ABFOPT1 logical or environment variable.

Default Image Name

Specifies the default name created by ABF for the executable image file. You can change this name when you create the image. See Creating an Image of the Application (see page 429).

Default Start Frame or Default Start Procedure

Specifies the name of the frame or procedure with which the application begins executing. This name must follow the standard object naming conventions. You can fill in only one of these fields.

If you set a default start frame and attempt to start from another frame, ABF prompts you to decide whether to override the default. See Set a Default Start Frame (see page 361) for more information.

Application Role

Specifies the role name under which the application runs. The role must be an existing role identifier. Roles allow you to give the application permissions beyond those of the user running the application. You can override the role from the Application Defaults frame when you create an image.

If you fill in the Application Role field, ABF prompts you to give the role's password when you test the application or create an executable image. The role password is not saved anywhere in the database. ABF remembers the password from the previous session only; if a reused password is accepted, the user receives a simple confirmation window.

This field is part of the Knowledge Management Extension. For more information on using application roles, see Roles for Applications (see page 537). For details on creating and using roles, see the *SQL Reference Guide*.

Always use compiled forms

Use this field to specify how the application behaves when run as an image. Enter yes or no to specify whether the application must use compiled forms for all Report and 4GL frames. The default is yes.

Usually, the best answer is yes. This allows forms to be activated faster and makes it easier to run the image against another database. If the form appears to be out of date when you build the image, ABF compiles the form.

If you answer no:

- Another field appears on the Application Defaults frame. This field asks whether new frames must use compiled forms by default. You are given the choice of converting all existing frames to the new default.
- Another field also appears on the Edit frames for all of the application's 4GL frames. This field allows you to specify, for each frame, whether it must use a compiled form.

The menu operations are the standard operations OK, Cancel, and Help.

OK

Keeps changes and return to previous frame

Cancel

Ignores changes and return to previous frame

Help

Accesses the Help Utility for this frame

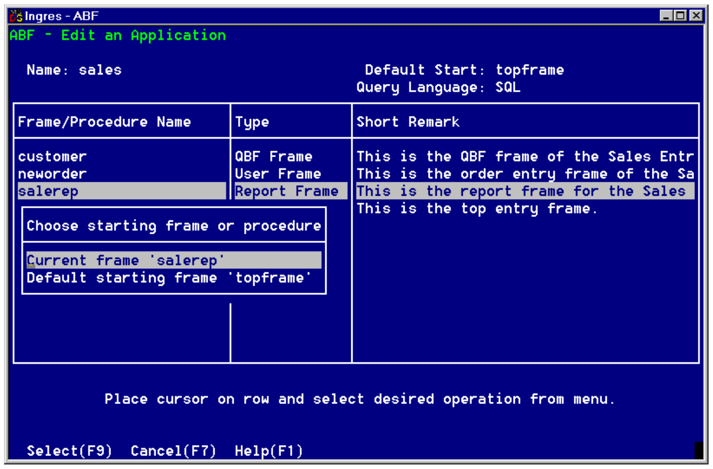
Set a Default Start Frame

If you want the application to automatically start running at a particular frame or procedure, set this up from the Application Defaults frame.

To set a default start frame

1. Select Defaults from the MoreInfo About an Application or the Edit an Application frame. This displays the Application Defaults frame, shown in the previous frame.
2. In the Default start frame field, enter the frame name and select OK.

If you set a default frame, and then select Go while another frame is highlighted, you see the pop-up shown in the following figure. Highlight your choice and choose Select. The application starts with the frame you choose.



```
Ingres - ABF
ABF - Edit an Application

  Name: sales                                 Default Start: topframe
                                             Query Language: SQL

 Frame/Procedure Name     Type           Short Remark

 customer                 QBF Frame      This is the QBF frame of the Sales Entr
 neworder                 User Frame     This is the order entry frame of the Sa
 salerep                  Report Frame   This is the report frame for the Sales
                                         This is the top entry frame.
  Choose starting frame or procedure

  Current frame 'salerep'
  Default starting frame 'topframe'


             Place cursor on row and select desired operation from menu.

  Select(F9)  Cancel(F7)  Help(F1)
```

Application Utilities Menu

The Application Utilities submenu is available from the Applications Catalog and the Edit an Application frame menus. When you select this option, the Utilities submenu takes the place of the frame menu on the menu line.

Image

Creates an executable version of the application to run from the operating system. See Creating an Image of the Application (see page 429) for more information.

AppReport

Displays the Application Report frame, where you can generate a summary report on the application which you can save for later use

Ingres/Menu

Calls Ingres Menu, from which you can call any Ingres Tool (including an interactive query language) except ABF without leaving the current ABF session. Exit returns you to ABF. You cannot use Ingres Menu to switch to another database.

VMS: To use Ingres Menu, set your VMS subprocess quota to at least 3.

Shell

UNIX, Windows: Escapes to the operating system prompt.

UNIX: This is the setting of the environment variable $SHELL. If $SHELL is not set, escape to the Bourne shell (/bin/sh). Exit returns you to ABF.

Spawn

VMS: Escapes to spawn new DCL process. At exit, return to ABF.

LockStatus

Displays the Lock Administration frame to review or change an application's lock status

Cleanup

Reviews and clean up database system tables. Available to DBA only.

Destroying an application or frame does not remove all of the information about the destroyed object from the extended system catalogs. This unused data can affect performance, but otherwise causes no loss of functionality. You can create a new frame or application using the same name as a deleted one.

To remove this unused data, use the Cleanup operation or run the sysmod utility.

ConstantsFile

Displays the Specify a Global Constants File frame to select an alternate constants file to be used in the application. From this frame, you can also create a constants file based on the default constants or call the system editor to edit a constants file. See Using Alternate Sets of Global Constant Values (see page 407) for more information.

Help, End

Perform standard operations

Call the Tables Utility

As part of the process of creating an application, you must decide on the elements of data you need and establish them in tables. If the database you are using already contains all the data you need, add more tables.

You can use the ABF Application Utilities menu to access the Ingres Tables Utility while you define frames or procedures for the application. You cannot create a new database at this time, however. See *Character-based Querying* and *Reporting Tools User Guide* for instructions on how to use the Tables Utility to create new tables or examine existing tables.

To call the Tables Utility

1. Choose Utilities from the Applications Catalog or the Edit an Application frame.
2. Choose Ingres/Menu.
3. Choose Tables.

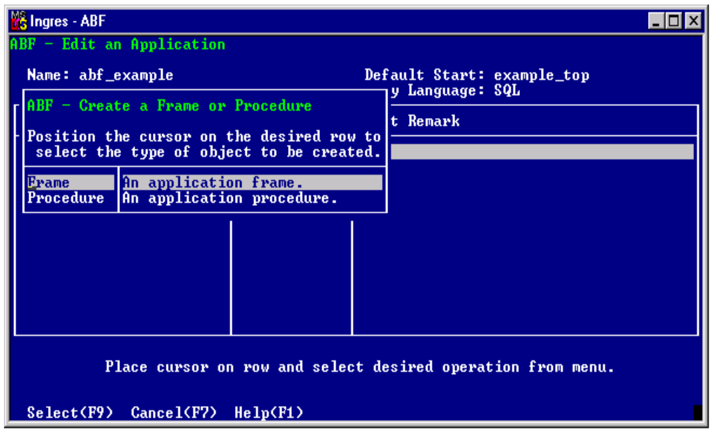
   The Tables Catalog frame appears, listing the tables in the database.
4. To leave the Tables Utility, select End on the Examine a Table frame, the Tables Catalog frame, and the Ingres Menu. You return to the Applications Catalog or the Edit an Application frame.

Note: Visual DBA provides an alternate way of creating and managing tables.

The Create a Frame or Procedure Frame

The frames are the points of interaction between the user and the application. You can assemble the frames when all the necessary tables are complete. If you must complete or check on tables, see Call the Tables Utility (see page 363).

The procedure for defining a frame is the same for all frame types, but you specify different details for each. Display the Create a Frame or Procedure pop-up frame, shown below, by selecting Create from the Edit an Application frame menu. It does not matter whether a line of the table field is highlighted.


```
Ingres - ABF
ABF - Edit an Application

  Name: abf_example                       Default Start: example_top
  ABF - Create a Frame or Procedure     y Language: SQL
                                        t Remark
  Position the cursor on the desired row to
   select the type of object to be created.

  Frame       An application frame.
  Procedure   An application procedure.

           Place cursor on row and select desired operation from menu.

  Select(F9)  Cancel(F7)  Help(F1)
```

This pop-up frame offers the following choices:

Frame

Creates a frame for the application

Procedure

Creates a procedure for the application

When the Create a Frame or Procedure pop-up is displayed, the menu operations on the Edit an Application frame change to the following:

Select

Selects current row as the component to create

Cancel

Cancels the operation and return to the previous frame

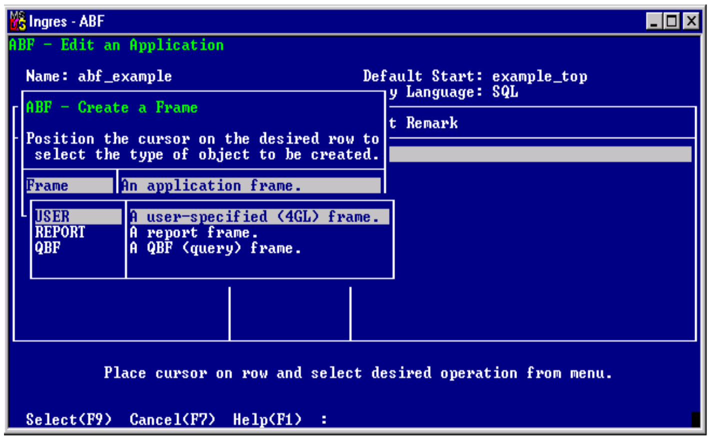
Help

Performs standard operation

Selecting Frame as the type of component to create displays the Create a Frame pop-up, described in the next section. Selecting Procedure displays the Create a Procedure pop-up, described in a subsequent section.

The Create a Frame Menu

Select Frame from the Create a Frame or Procedure pop-up frame to display the Frame menu, shown in the following figure:



```
Ingres - ABF
ABF - Edit an Application

 Name: abf_example                       Default Start: example_top
                                        y Language: SQL
 ABF - Create a Frame
                                        t Remark
 Position the cursor on the desired row to
  select the type of object to be created.

 Frame      An application frame.

  USER       A user-specified (4GL) frame.
  REPORT     A report frame.
  QBF        A QBF (query) frame.


           Place cursor on row and select desired operation from menu.

 Select(F9)  Cancel(F7)  Help(F1)  :
```

This lists the types of frames you can create:

USER

Specifies a user-specified (4GL) frame

REPORT

Specifies a report frame

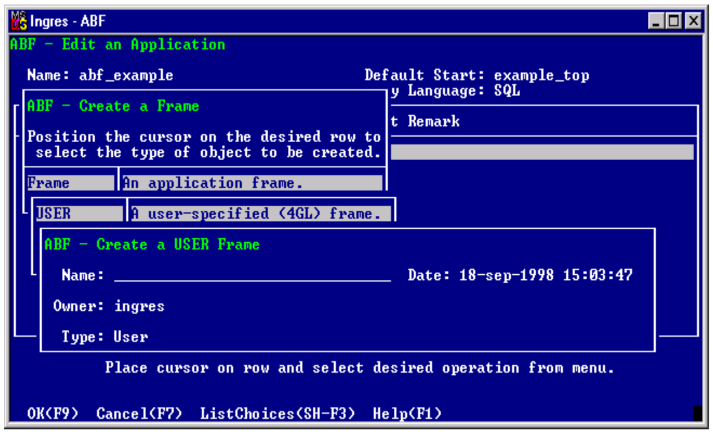
QBF

Specifies a QBF (query) frame

Selecting any of the options leads to a pop-up Create a *frametype* Frame, where *frametype* is the selected type.

Create a Frametype Frame Pop-up

The first step in creating a frame is naming the frame through the Create a *frametype* Frame pop-up. The *frametype* is USER, REPORT, or QBF. These four pop-up frames are identical except for the *frametype* in the pop-up's title. The following figure shows an example of the Create a *frametype* Frame pop-up, where USER is the *frametype*:



```
Ingres - ABF
ABF - Edit an Application

  Name: abf_example                         Default Start: example_top
                                            y Language: SQL
  ABF - Create a Frame
                                            t Remark
  Position the cursor on the desired row to
   select the type of object to be created.

  Frame      An application frame.

   USER       A user-specified (4GL) frame.
    ABF - Create a USER Frame

      Name: ________________________________   Date: 18-sep-1998 15:03:47

     Owner: ingres

      Type: User

           Place cursor on row and select desired operation from menu.

  OK(F9)  Cancel(F7)  ListChoices(SH-F3)  Help(F1)
```

The Create a Frame pop-up has four fields. This first field is the only one you can fill in:

Name

Specifies a name field in which you enter the name of the frame. The name must follow standard object naming conventions.

The other three fields contain read-only defaults:

Owner

Specifies the name of the user who created the frame

Type

Specifies the frame type selected in the Create a Frame or Procedure pop-up

Date

Specifies the creation date of the frame

The underlying Edit an Application frame provides the following menu operations for the Create a Frame pop-up:

OK

Creates the frame and displays the Edit a Frametype Frame Definition frame for the frame type you selected

Cancel

Cancels the operation and return to the Applications Catalog frame

ListChoices

Displays a list of choices for the current field from which you can select or a field description

Help

Accesses the Help Utility for this frame

The following sections introduce each frame type shown on the Create a Frame or Procedure Frame.

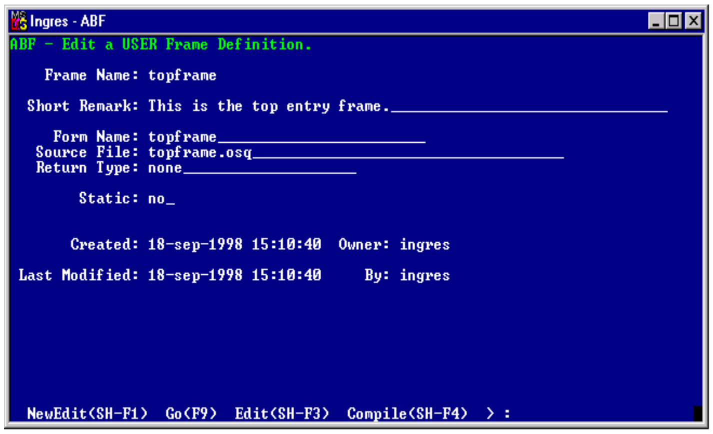
Creating a User-Specified Frame

A user-specified or USER frame has two basic components, although you can include others:

- A form created with the ABF FormEdit operation
- A 4GL source code file that you enter using the system editor

In addition, you can create procedures, variables, and constants that the source code file can call and manipulate. Creating procedures, variables, and constants is discussed below.

The Edit a USER Frame Definition frame is equivalent to the Edit an Application frame for applications. Define the basic details for the user frame here. From this frame, use the FormEdit operation to create the form. Use the Edit operation to call the system editor to create the source code file.



```
ABF - Edit a USER Frame Definition.

    Frame Name: topframe

  Short Remark: This is the top entry frame._______________________________

     Form Name: topframe________________________
   Source File: topframe.osq____________________________________
   Return Type: none____________________

        Static: no_

       Created: 18-sep-1998 15:10:40  Owner: ingres

 Last Modified: 18-sep-1998 15:10:40     By: ingres




  NewEdit(SH-F1)  Go(F9)  Edit(SH-F3)  Compile(SH-F4)  > :
```

You can enter and edit the Short Remark field, Form Name, Source File, Return Type, and Static fields here.

Frame Name

(Read-only) Holds the name of the frame

Short Remark

Specifies an optional brief description of the frame

Form Name

Specifies the form associated with this frame. The default is the frame name, but you can change this. For more information on creating forms, see Forms in ABF Applications (see page 505).

Source File

Specifies the name of the 4GL source code file associated with this frame. The default filename combines the suffix .osq with the frame name. The suffix .osl appears with QUEL files. For information on using QUEL, see Notes for Users of QUEL (see page 1263).

The file name must be unique to the frame—it cannot belong to another frame or procedure. ABF can truncate or change the default name if it is too long or conflicts with other source code file names. Use the Edit operation to create the file with the system editor. Do not include a directory path as part of the file name. All source code files must reside in the specified source code directory.

Return Type

Provides the data type of any values to be returned from this frame.

Legal return types are any Ingres type, "string," or "none." You cannot use record or array types. To see available Return Types, use the ListChoices menu operation.

A "string" or Ingres type can also be nullable. Use the Nullable field to specify this.

Static

Indicates yes or no (the default) to specify whether the data on the frame's form is saved and redisplayed the next time the frame is called.

- Typically, this field must be set to no. If you enter no, ABF clears the fields on the frame's form each time the application displays the frame.
- If you enter yes, ABF preserves the data entered on the frame's form, and redisplays the form with that data the next time the application calls the frame.

  If a form is shared by more than one frame, do not use the Static feature. Data entered on the Static form in one frame appears in the other frame.

If the current application uses some non-compiled forms, another field appears that allows you to specify whether this frame uses a compiled form or fetches its form from the database at run time.

The remaining fields give information about the user who created the form (the Owner), and when the frame was created and its definition last modified.

These operations are available from the Edit a USER Frame Definition frame:

NewEdit

Selects or creates the definition of a different component from a submenu that appears on the menu line. Prompts for type and name of component.

Go

Runs the application starting at the current frame. You cannot run the application from an individual frame unless all the frames are error-free.

Edit

Uses the system editor to create or edit the file containing the 4GL specification for the current frame

Compile

Checks the 4GL syntax for current frame, display errors (if any), and add the compiled source code for the frame to the application if free of errors. If errors exist, ABF creates an error listing file in the object code directory. ABF asks if you want to display the errors.

FormEdit

Runs Visual Forms Editor (VIFRED) to create or edit a form for the current frame

LongRemark

Displays or enters more information about the selected frame on the LongRemark pop-up frame

Print

Prints the frame definition including the text file containing the 4GL specification

ListChoices

Displays a menu of legal values for the current field *or* a field description

Cancel, Help, End

Perform\ standard operations

Create a User-Specified Frame

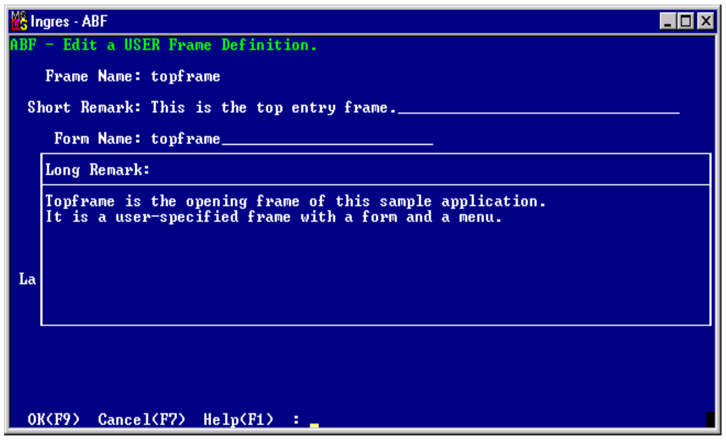
These steps take you through the windows and pop-ups involved in creating a user-specified frame, referring you to the figure that shows each frame. Use the ListChoices menu operation to see a list of legal values from which you can choose *or* a field description for each of these fields.

To create a user-specified frame

1. Choose Create from the Edit an Application frame. You see the Create a Frame or Procedure pop-up, as illustrated.
2. Highlight Frame and press Return or choose Select from the menu. The Create a Frame options pop-up appears, as illustrated, listing the available types of frames.
3. Select USER. The Create a USER Frame pop-up appears with a blank Name field. The Owner, frame Type and Date of creation are filled in, as illustrated.
4. In the Name field, enter the name for the frame. The other fields are display only.

5. Choose OK. ABF creates the frame and displays the Edit a USER Frame definition frame, as shown in the example.
6. In the Form Name field, enter the name of the form to be associated with the frame. By default the form has the same name as the frame.
7. In the optional Short Remark field, enter a brief description of the frame if you like.
8. In the Source File field, enter the filename you are going to use when you enter the 4GL code or the name of an existing file. The default filename combines the suffix .osq with the frame name.
9. In the Return Type field, enter the data type for data this frame returns to a frame or procedure that calls it. The default value for the Return Type field is "none."

   For the data types that you can use here, see the 4GL part of this guide.
10. Enter yes or no (the default) in the Static field. For details on using the Static field, see the Creating a User-Specified Frame section.



```
ABF - Edit a USER Frame Definition.

    Frame Name: topframe

  Short Remark: This is the top entry frame.________________________________

     Form Name: topframe________________________
    Long Remark:

    Topframe is the opening frame of this sample application.
    It is a user-specified frame with a form and a menu.

  La

  OK(F9)  Cancel(F7)  Help(F1)  : _
```

11. Select LongRemark to enter or view more detailed information about the application on the LongRemark pop-up, shown above. The other Edit a *Frametype* Frame Definition frames also have associated Long Remark pop-ups.

This completes the creation of a user-specified frame. Now you can go on to create a form for this frame.

Create a Form

The following steps describe using the ABF FormEdit operation to create a form for a user-specified frame.

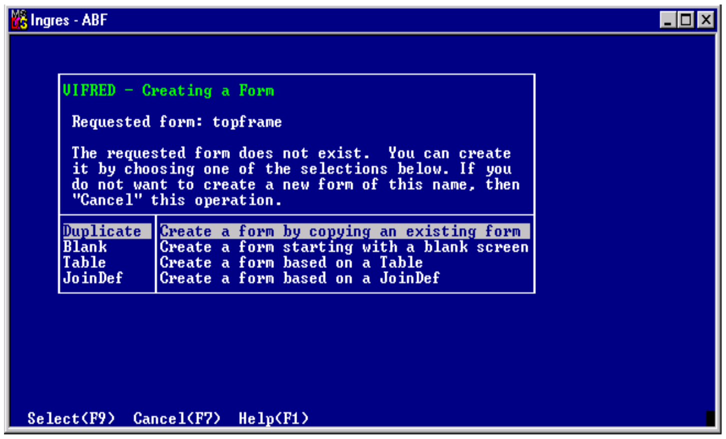
To create a form

1. Complete the fields in the Edit a USER Frame. Enter a form name, such as "topframe," in the Form Name field.

2. Choose the FormEdit operation to enter VIFRED. The following message appears:

   ```
   Retrieving form 'topframe' .  .  .
   ```

   Because this form does not yet exist, the VIFRED Creating a Form frame appears, as shown below. The frame menu lists four default forms: Duplicate, Blank, Table, and JoinDef.

   ```
   VIFRED - Creating a Form

    Requested form: topframe

    The requested form does not exist.  You can create
    it by choosing one of the selections below. If you
    do not want to create a new form of this name, then
    "Cancel" this operation.

   Duplicate   Create a form by copying an existing form
   Blank       Create a form starting with a blank screen
   Table       Create a form based on a Table
   JoinDef     Create a form based on a JoinDef

     Select(F9)  Cancel(F7)  Help(F1)
   ```

3. Choose a default form for the frame you are creating:

   Duplicate

   Copies an existing form

   Blank

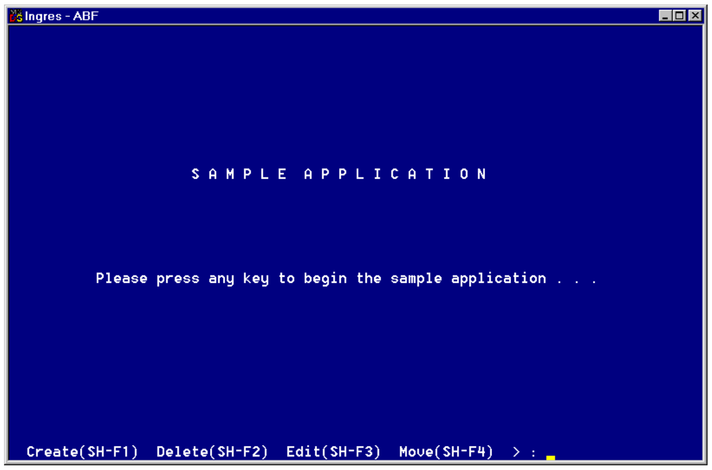
   Starts with a blank form. This is useful for menu forms and forms which are not based on a single table. The following figure shows an example.

Table

Creates a form that displays all or most of the columns in a table. This option provides a default form that you can modify.

JoinDef

Creates a form that displays all or most of the columns in a JoinDef. This option provides a default form that you can modify.



```
S A M P L E   A P P L I C A T I O N

Please press any key to begin the sample application . . .

Create(SH-F1)  Delete(SH-F2)  Edit(SH-F3)  Move(SH-F4)  > :
```

4. Use VIFRED operations to design the form, creating the required fields.

   Developers writing applications that are used with mouse support must keep in mind that the mouse user can click at random around the fields of a form. This affects the way you set up fields and their validations in VIFRED. For more information about form design in VIFRED, see the *Using Character-based Querying and Reporting Tools Guide.*

5. When you finish designing the form, choose Save to write the form to the database. You see the VIFRED Saving a Form frame. Choose Save and End to return to the Edit A USER Frame Definition frame.

The next step is to create the frame menu, in the form of a source code file.

Create the 4GL Source Code File for a User-Specified Frame

A unique feature of a user-specified frame is that you can create a 4GL source code file which provides the menu and operations for the frame. In other Ingres tools, such as RBF, the frame uses the default menu for the interface.

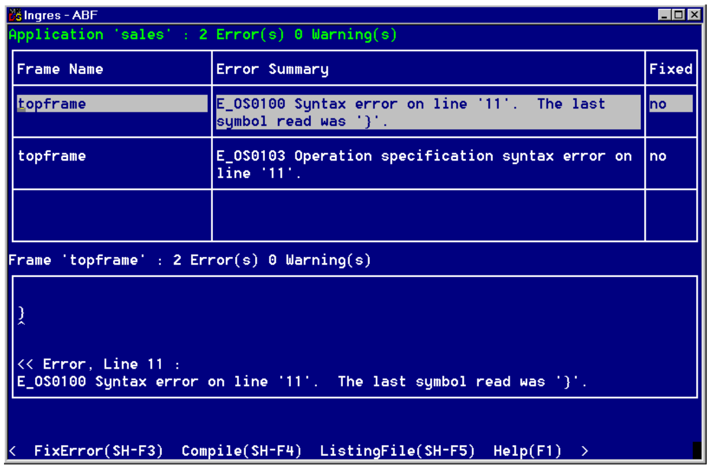
The 4GL code for a user frame is stored in a text file in the source code directory. The source file must be unique and cannot be shared between frames.

The 4GL part of this guide provides detailed information on writing 4GL specifications.

To create a source code file containing 4GL statements

1. Choose Edit from the Edit a USER Frame Definition frame. ABF runs the editor specified with ING_EDIT or the default system editor. (You can use the Edit operation later to edit existing source files.)
2. Your system editor's opening display appears. Follow the usual text-editing procedures to enter the 4GL code.
3. When you finish creating the file, save it before you exit the editor. You return to the Edit a USER Frame Definition frame.
4. Compile the frame. This places a compiled copy of the 4GL code in the database. Choose Compile to compile the code and check for syntax errors. You see a message telling you that the frame is being compiled.
5. If there are compilation errors, ABF displays a frame which lists the errors and prompts you to correct them. Enter y (yes) to see the Error Listing frame, shown in the following figure.

   This frame displays any error messages in a scrollable table field. Select FixError to open the file. Select MarkFixed to mark an error corrected. Select Compile again when finished.



```
Application 'sales' : 2 Error(s) 0 Warning(s)
```

Frame Name	Error Summary	Fixed
topframe	E_OS0100 Syntax error on line '11'. The last symbol read was '}'.	no
topframe	E_OS0103 Operation specification syntax error on line '11'.	no

```
Frame 'topframe' : 2 Error(s) 0 Warning(s)

}
^

<< Error, Line 11 :
E_OS0100 Syntax error on line '11'.  The last symbol read was '}'.
```

Ingres - ABF
< FixError(SH-F3) Compile(SH-F4) ListingFile(SH-F5) Help(F1) >

6. You can print a copy of the 4GL specification on your printer by choosing Print from the Edit a USER Frame Definition frame.
7. When compilation is complete, choose End to return to the Edit an Application frame.

Adding a New Frame to an Application Menu Frame

If your application has a menu frame, you can add a new frame to the menu after it is complete by opening the Edit a USER Frame for the menu and following these steps:

- Add the 4GL code for the new frame to the code for the main menu. To do this, choose the Edit operation to open the source code file.
- Edit the main menu form to list the new frame. To do this, choose FormEdit to enter VIFRED and modify the form.

The new frame appears on the menu line when you run the frame next.

Testing the USER Frame

The last step in creating a frame is testing it.

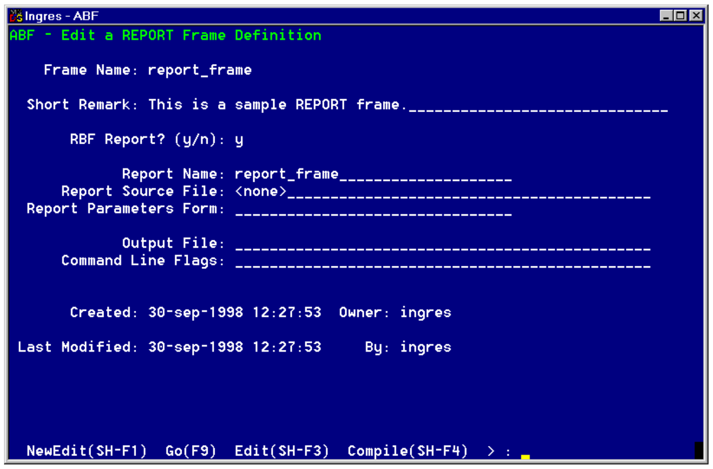
1. To check to see that your USER frame is operating in the way you intend, use the Go operation from the Edit an Application frame. This compiles the application, and, if no errors prevent it from running, runs it.
2. If you are not satisfied with the way your frame is functioning, return to the frame and use Edit or FormEdit to correct the source code or the form.
3. Select Go to run the application again. For more information on Go, see the Testing Incomplete Applications section.

Creating Report Frames

The next option on the Edit an Application frame's Create a Frame or Procedure pop-up is REPORT. Defining Report frames is similar to defining USER frames.

Select REPORT from the Create a Frame or Procedure frame. Specify the name in the Create a Report frame and select OK. The Edit a REPORT Frame Definition frame appears, as in the following figure.

In creating a report frame, you enter the Ingres Reports interface. To learn more about this interface, see the *Using Character-based Querying and Reporting Tools*.



```
ABF - Edit a REPORT Frame Definition

    Frame Name: report_frame

  Short Remark: This is a sample REPORT frame._______________________________

       RBF Report? (y/n): y

             Report Name: report_frame____________________
      Report Source File: <none>_________________________________________
  Report Parameters Form: ________________________________

             Output File: ________________________________________________
      Command Line Flags: ________________________________________________


       Created: 30-sep-1998 12:27:53  Owner: ingres

 Last Modified: 30-sep-1998 12:27:53     By: ingres



  NewEdit(SH-F1)  Go(F9)  Edit(SH-F3)  Compile(SH-F4)  > :
```

The fields on this frame contain the information needed to produce reports from the Report interface.

Frame Name

Specifies the name of the report frame

Short Remark

Specifies a brief description of report frame

RBF Report? (y/n)

Determines whether the report is a Report-Writer or RBF report

Report Name

Specifies the name of an existing report you plan to use as is or edit for use in this frame. Or, if you are creating a new report, enter the name here. If you are using Report-Writer, this name appears in the Report-Writer specification file for the report.

Report Source File

Specifies the name of the report specification file for a Report-Writer report. File names specified here require an extension of .rw. RBF reports do not have a report specification file.

Report Parameters Form

Specifies the name of a form on which the user can enter values at run time for any variables used in the report. Variable values entered on this form are passed as parameters to the Report-Writer at run time.

To create or edit a form, specify the form name in this field, and choose the FormEdit operation.

When you create a parameter form, the names of the fields on the form must match the variable names in the report specification file. For example, if *year* is a variable used in the where clause of a query "...where pop. year = $*year*" then the variable form must include the field *year.*

The Report-Writer prompts the application user for needed values if they are not passed. It is strongly recommended that all variables used in a report be on the report frame's form. See *Using Character-based Querying and Reporting Tools* for more information on variables.

Output File

Specifies the report's output type: file, print, or display (the default).

- To store the report in a file, enter the *filename* to receive the report.
- To send the report to the printer, type printer.
- To display the report on your screen, type terminal. This is the default; if you leave this field blank, the report is directed to your terminal screen.

You can also save output in a file by using the .output command in your report specification file. However, any file name you enter on the Edit a REPORT Frame Definition frame overrides a file name given in the .output command.

Command Line Flags

Specifies the report parameters you want the application to use when running the report. Running a report from an application is like specifying the report command with the -r parameter at the operating system:

```
report dbname -r reportname
```

You can specify additional parameters for the report command in the Command Line Flags field. For example, specify
-u*username* to run the report as another user, or **-**l*pagewidth* to set the maximum page width. You cannot specify a parameter, such as -m or -i that conflicts with the -r parameter. For information on using parameters with the report command, see the *Character-based Querying and Reporting Tools User Guide.*

The following menu operations are similar to those in the Edit a USER frame:

NewEdit

Edits or creates a different frame or procedure. Prompts the user for the type of object to define.

Go

Runs the report frame

Edit

Runs the report editor you specified for the report in the RBF Report? (y/n): field.

- If you answered yes, *RBF* is the editor.
- If you answered no, you use the system editor.

Compile

Calls sreport to compile the source code for the Report-Writer report. This option is not necessary for RBF reports.

FormEdit

Runs VIFRED to create or edit a form for the current frame. Not active unless you specify a form.

LongRemark

Displays or edits more information about the selected frame on the LongRemark pop-up

Print

Prints a description for the frame

ListChoices, Cancel, Help, End

Perform standard operations

Create a Report

To create a REPORT frame for your application

1. From the Edit an Application frame, choose Create. This displays the Create a Frame or Procedure pop-up, as illustrated.
2. Select Frame. You see the pop-up menu for the type of frame.
3. Select REPORT to display the Create a REPORT Frame pop-up. This frame operates in the same way as the Create a USER frame pop-up.
4. In the Name field, type the name for the report frame, and select OK.

5. You see the Edit a REPORT Frame Definition frame, with the default frame and report names filled in, as illustrated in the example.
6. (Optional) Enter a brief remark in the Short Remark field.
7. In the RBF Report? Field, enter y (the default) to create an RBF report, or n to create a Report-Writer report.
8. In the Report Name field, either take the default name (the same name as the report frame) or enter the name of an existing or new report.
9. The next field is the Report Source File field.
   - *If you entered n in the RBF Report? Field,* type the name of the report specification file in the Report Source File field. This filename requires an extension of .rw. Proceed to step 10.
   - *If you entered y in the RBF Report? Field,* the Report Source File field now reads <none> as RBF reports do not require source code files. Proceed to step 11.
10. For Report-Writer reports only: When you are ready to create the report specification file, use the Edit operation to enter the system editor.

    Type in the file and save it using system editor commands. Save the report specification file in the database by using the Compile operation. For a sample source code file, see the Sample Report-Writer Report File section.

    Alternatively, you can create and save a Report-Writer report outside of ABF using the system editor and sreport.
11. For either the RBF or Report-Writer report, you can optionally create a form for the entry of values for variables used in the report at run time.
12. In the Report Parameters Form field, type the name of the form. For details, see the Creating Report Frames section.

    When you use RBF to build reports, you can specify variables for reports in the Column Options frame of RBF. For details, see the sections on RBF in *Character-based Querying and Reporting Tools user Guide.*
13. Select the output type next. In the Output File field, enter a filename, type printer, or type terminal to display the report on your terminal screen (the default).

    These output options are discussed in the Creating Report Frames section.
14. Use the Command Line Flags field to enter any parameters you want the report to use when running.
15. If you are creating a report with RBF, select Edit and use RBF in the normal manner. This procedure does not produce a stored report or report source file.

For additional information about creating reports, see the *Character-based Querying* and *Reporting Tools User Guide.*

Sample Report-Writer Report File

This sample report source file takes data from the Emp database table and formats it, showing the salaries of employees and the total salary for each department. In the report, *empdetail.rw* is the name of the report source file, while *empdetail.out* is the name of the report output file.

```
.name empdetail.rw
.output empdetail.out
.data emp
.sort dept : a,
           name : a

.header report
     .newline 3
     .center
     .print "List of Employees"
     .newline 3

.header dept
     .tformat dept (c12)
.footer dept
     .right salary
     .print     "_________________"
     .tab salary
     .print sum(salary)
     .newline 2

.detail
     .print dept (b12), name (c14)
     .print date(birthdate)
          (d"February | 3, 1901 ")
     .println salary ("$$,$$$,$$$.nn")
```

Testing the Report Frame

The last step in creating a frame is testing. You can test a Report frame without compiling the application by choosing Go from the Report frame itself.

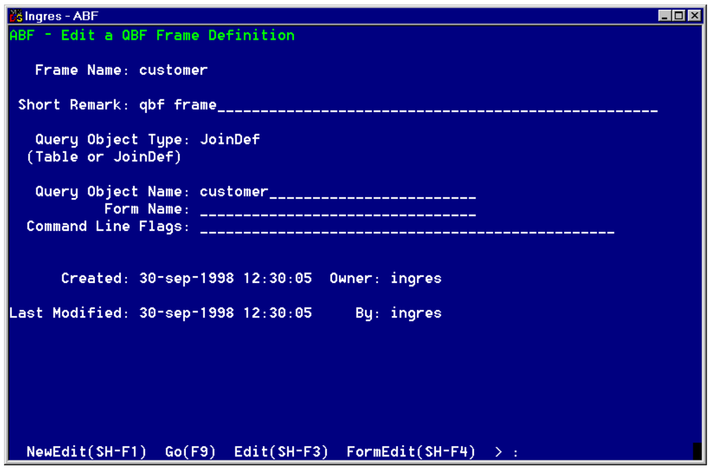
You can also test the entire application at any time during development by selecting Go from the Edit an Application frame. For more information, see Testing Incomplete Applications (see page 426).

Creating a QBF Frame

The next option on the Create a Frame or Procedure pop-up frame is the QBF Frame. A QBF frame consists of:

- A database query using the QBF user interface.
- A default form or a form created using the ABF FormEdit operation.

No programming with 4GL is necessary. The Edit a QBF Frame Definition frame is shown in the following figure:



```
ABF - Edit a QBF Frame Definition

   Frame Name: customer

 Short Remark: qbf frame___________________________________________________

   Query Object Type: JoinDef
  (Table or JoinDef)

   Query Object Name: customer________________________
           Form Name: ________________________________
  Command Line Flags: ________________________________________________


      Created: 30-sep-1998 12:30:05  Owner: ingres

Last Modified: 30-sep-1998 12:30:05     By: ingres



  NewEdit(SH-F1)  Go(F9)  Edit(SH-F3)  FormEdit(SH-F4)  > :
```

In this frame, you enter the values ABF needs to run QBF. The fields are similar to those on the Edit a USER and Edit a REPORT frames.

Frame Name

Specifies the name of the query target; this name must follow standard object naming conventions.

Short Remark

(Optional) Specifies a brief description of the frame

Query Object Type

Specifies the values are either Table or JoinDef (the default); this name must follow standard object naming conventions

Query Object Name

Specifies the name of database table or JoinDef; this name must follow standard object naming conventions. The default is the frame name.

Form Name

Specifies the name of the form associated with this frame; this name must follow standard object naming conventions. The value is the name of a form or the default (blank), which generates a default form when QBF is run.

Command Line Flags

Specifies the parameters that define the scope of the query.

The only parameter available on this frame is –m. Use –m with an additional parameter to indicate the mode of the form. For example, using –mretrieve automatically puts the form in retrieve mode and not give the user access to update or append mode. If you do not use this parameter, the user must select the form mode from the QBF Execution Phase frame menu.

The following menu operations are similar to those in the Edit a USER or Report frame:

NewEdit

Edits or creates a different frame or procedure. You are prompted for the type of component.

Go

Tests the frame by running QBF on the named Table or JoinDef

Edit

Edits, views, or creates the query or JoinDef:

- *If the Query Object Type is a Table*, Edit starts the Tables Utility, displaying a list of database tables.
- *If the Query Object Type is a JoinDef*, Edit starts QBF and goes directly to the Edit a JoinDef frame.

FormEdit

Runs VIFRED to create or edit a form for the current frame. You must specify a form name before using FormEdit.

LongRemark

Displays or edits more information about the frame on the LongRemark pop-up

Print

Prints the definition of the frame

ListChoices, Cancel, Help, End

Perform standard operations

Create a QBF Frame

To create a QBF frame:

1. From the Edit an Application frame, choose Create.
2. Select Frame from the Create a Frame or Procedure pop-up. The Create a Frame or Procedure pop-up figure shows this stage.

3. Select QBF. The Create a QBF Frame pop-up appears. It is identical in format to the Create a USER Frame pop-up.
4. In the Name field enter the name for this QBF frame. Choose OK.
5. The Edit a QBF Frame Definition frame appears, showing the name you selected in Step 3 as the Frame Name and the default Query Name.

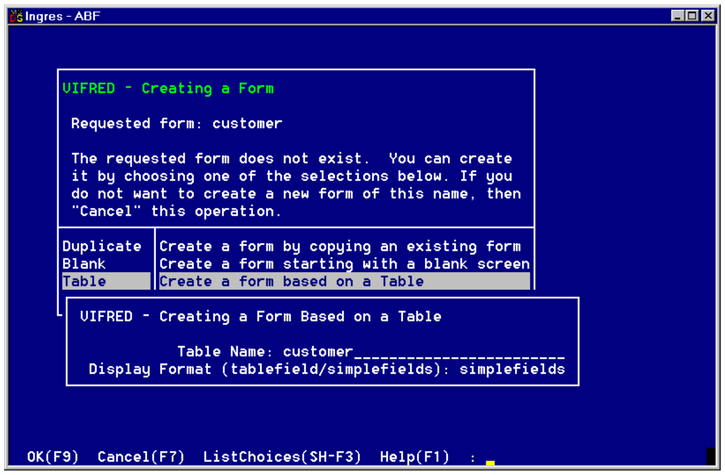
   Optionally enter a description in the Short Remark field.
6. In the Query Object Name field, type the name of the query target. Indicate whether this object is a table or JoinDef in the Query Object Type field.
7. In the Form Name field, enter a form name. You can specify a form that already exists or create a new one as described below.
8. In the Command Line Flags line, optionally enter parameters for runtime operation of the frame.

Create a Form for the QBF Frame

You can create a form with VIFRED before creating the QBF frame, or you can call VIFRED to create a form from the Edit a QBF Frame Definition frame.

To create a form from the Edit a QBF Frame Definition frame

1. Specify a form name on the Form Name field.
2. Choose FormEdit from the menu on the Edit a QBF Frame Definition frame. If the form does not exist, VIFRED displays the Creating a Form frame.



```
VIFRED - Creating a Form

 Requested form: customer

 The requested form does not exist.  You can create
 it by choosing one of the selections below. If you
 do not want to create a new form of this name, then
 "Cancel" this operation.

Duplicate  Create a form by copying an existing form
Blank      Create a form starting with a blank screen
Table      Create a form based on a Table

  VIFRED - Creating a Form Based on a Table

              Table Name: customer_________________________
   Display Format (tablefield/simplefields): simplefields

OK(F9)  Cancel(F7)  ListChoices(SH-F3)  Help(F1)  :
```

3. Select one of the form options.

   If you specify Blank, ABF places you in VIFRED and displays a blank form. Proceed to step 6.

   If you specify Duplicate, Table, or JoinDef, ABF displays a pop-up Creating a Form Based on a *Formtype*. The figure above shows the pop-up Creating a Form Based on a *Formtype* where the *formtype* is the Table option.

   The pop-up operates in a similar way for each option.

4. Enter the name of a table, a JoinDef, or an existing form to duplicate. Type in the name, or use the ListChoices operation to see a menu-type list of available tables, JoinDefs, or forms.

   If you are creating a form based on a table, enter tablefield or simplefields in the Display Format field.

5. Select OK.

   When you edit or create a form for a QBF frame, it is not necessary to specify a QBFName, because ABF sets up the form and the query target in your frame specification. See the *Character-based Querying and Reporting Tools User Guide* for a description of QBFNames.

6. Create the form by moving elements or creating new ones using VIFRED commands, then choose Save. VIFRED asks you to verify the name under which you are saving the form, and then returns you to the Edit a QBF Frame Definition frame.

If you are creating an application that uses a mouse, remember that the mouse user can click at random around the fields of a form. This affects the way you set up fields and their validations in VIFRED. Form design in VIFRED is covered in more detail in *Character-based Querying and Reporting Tools User Guide.*

For more information about using QBF, see *Character-based Querying and Reporting Tools User Guide.*

Testing the QBF Frame

The last step in creating a frame is testing. You can test a QBF frame without compiling the application by choosing Go from the QBF frame itself.

You can also test the entire application at any time during development by selecting Go from the Edit an Application frame. For more information, see Testing Incomplete Applications (see page 426).

Creating Procedures

A *procedure* is a series of statements written in 4GL, Ingres SQL (called a database procedure), or 3GL. You define the procedure using ABF. You can include procedures anywhere in your application that uses a frame (including the start frame). When you call the procedure, ABF executes the commands it contains.

You can use the following types of procedures in ABF applications:

4GL procedures

A 4GL procedure can call any other type of procedure. The procedures must be declared to ABF. A 4GL procedure does not include forms and therefore cannot include statements that refer to fields on a form.

3GL procedures

A host language procedure can access forms or the database through embedded query language statements. A 3GL procedure can call other 3GL procedures or database procedures. As long as these internal procedures are not called directly from 4GL, it is not necessary to define them to ABF.

Place the code for these nested procedures in a file that ABF knows about or include the object module in the default link options list. It is always safer to define procedures to ABF, whether or not they are called directly from 4GL.

Database procedures

A database procedure is a series of Ingres SQL statements and is stored as an object in the database. A database procedure is a library-only procedure; it cannot call any of the other procedure types. Also, a database procedure cannot include any statements that refer to forms.

You cannot use database procedures in your applications if you want them to access non-Ingres databases through an Enterprise Access Product.

Examples—4GL procedures:

The first procedure, *addtax*, performs a frequently used tax calculation, then returns a result to the calling frame.

```
procedure addtax (cost = float8,
     taxrate = float4) =
     begin
          cost = cost + (cost * taxrate);
          return cost;
     end
```

The second procedure, *find*, returns a customer name from the database, given a customer number.

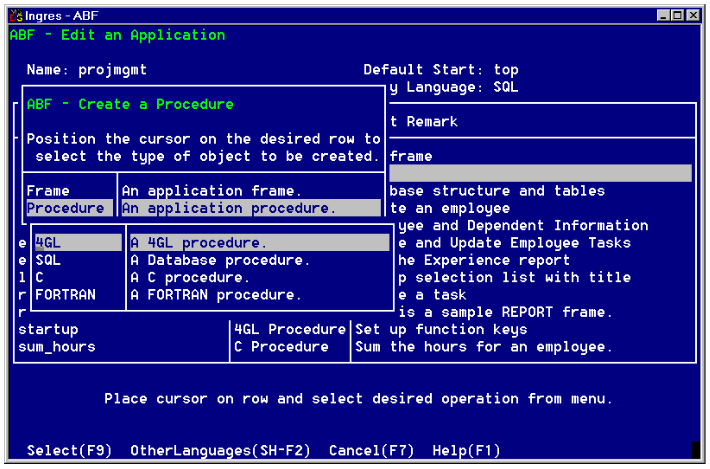
```
procedure find (cust_num = integer) =
     declare cust_name = char(31)
begin
     cust_name := '';
     find := select cust_name = name
          from customer
          where number = :cust_num;
     return cust_name;
end
```

How You Can Use a Procedure in an Application

1. Create the procedure using the ABF Edit an Application frame and the Create a Procedure frame for the procedure type you choose.
2. Write the code for the procedure as follows:
   - *For 4GL and 3GL procedures*, use the Edit operation of the Edit a Procedure Definition frame to call your default editor.
   - *For database procedures*, use the SQL create procedure statement.

   You must code database procedures outside of ABF. See the *SQL Reference Guide* for more information about creating database procedures.

Creating a procedure is similar to creating a frame. Select Procedure from the Create a Frame or Procedure pop-up frame of the Edit an Application frame. You see a menu of the types of procedures, as in the following figure:



```
Ingres - ABF
ABF - Edit an Application

  Name: projmgmt                               Default Start: top
                                               y Language: SQL
  ABF - Create a Procedure
                                               t Remark
  Position the cursor on the desired row to
   select the type of object to be created.    frame

  Frame      An application frame.             base structure and tables
  Procedure  An application procedure.         te an employee
                                                yee and Dependent Information
e  4GL        A 4GL procedure.                  e and Update Employee Tasks
e  SQL        A Database procedure.             he Experience report
l  C          A C procedure.                    p selection list with title
r  FORTRAN    A FORTRAN procedure.              e a task
r                                               is a sample REPORT frame.
startup                     4GL Procedure Set up function keys
sum_hours                   C Procedure   Sum the hours for an employee.

          Place cursor on row and select desired operation from menu.

  Select(F9)  OtherLanguages(SH-F2)  Cancel(F7)  Help(F1)
```

The following menu operations are available for this frame:

Select

Displays the Create a *Proceduretype* Procedure frame for the selected type of procedure

Other Languages

Displays a list of all callable non-embedded languages supported by your site

Cancel, Help

Perform standard operations

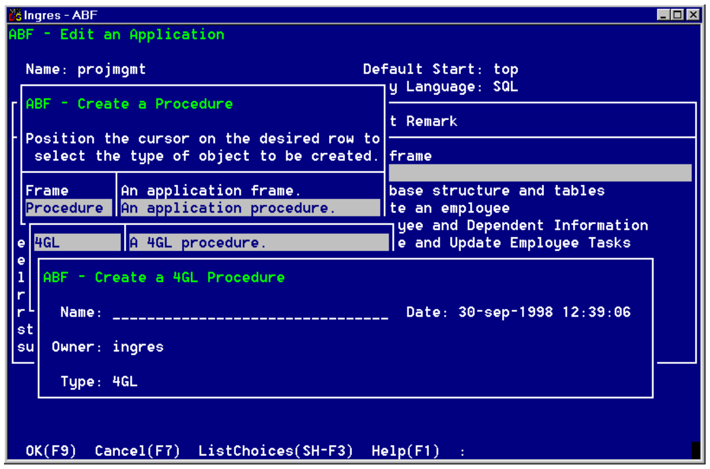
The procedure type you specify determines the source file extension, which in turn determines the compiler to be used. The programming language also correlates with the procedure type. ABF requires one of these extensions:

Operating System	SQL File Extension	non-SQL File Extension	Programming Language
Windows	.osq	none	4GL/SQL
	.sc	.c	Embedded SQL/C
	.scb	.cob	Embedded SQL/COBOL
UNIX	.osq	none	4GL/SQL
	.sc	.c	Embedded SQL/C
	.scb	.cob	Embedded SQL/COBOL
	.sf	.f	Embedded SQL/FORTRAN
VMS	.osq	none	4GL/SQL
	.sa	.ada	Embedded SQL/ADA
	.sc	.c	Embedded SQL/C
	.scb	.cob	Embedded SQL/COBOL
	.sf	.for	Embedded SQL/FORTRAN

Creating a Procedure Frame

After selecting Procedure from the Create a Frame or Procedure pop-up of the Edit an Application frame, you see a menu of procedure types. Any type you select leads to the pop-up Create a *Proceduretype* Procedure Frame. The name of the procedure you choose takes the place of *proceduretype* in the title.

The following figure shows the Create a Procedure frame for the 4GL procedure type:



```
Ingres - ABF
ABF - Edit an Application

  Name: projmgmt                               Default Start: top
                                              y Language: SQL
  ABF - Create a Procedure
                                              t Remark
  Position the cursor on the desired row to
   select the type of object to be created.   frame

  Frame       An application frame.           base structure and tables
  Procedure   An application procedure.       te an employee
                                               yee and Dependent Information
e  4GL         A 4GL procedure.                e and Update Employee Tasks
e
l   ABF - Create a 4GL Procedure
r
r     Name: ________________________________   Date: 30-sep-1998 12:39:06
st
su   Owner: ingres

      Type: 4GL


  OK(F9)  Cancel(F7)  ListChoices(SH-F3)  Help(F1)  :
```

The Create a Procedure frame has the same four fields for each of the procedure types:

Name

Specifies the name of the procedure

Owner, frame Type, and Date

Indicate read-only defaults, as shown in the preceding figure

The following menu operations are available from the Create a Procedure Frame:

OK

Creates the frame

Cancel, ListChoices, Help

Perform standard operations

Create a Procedure

To create a procedure

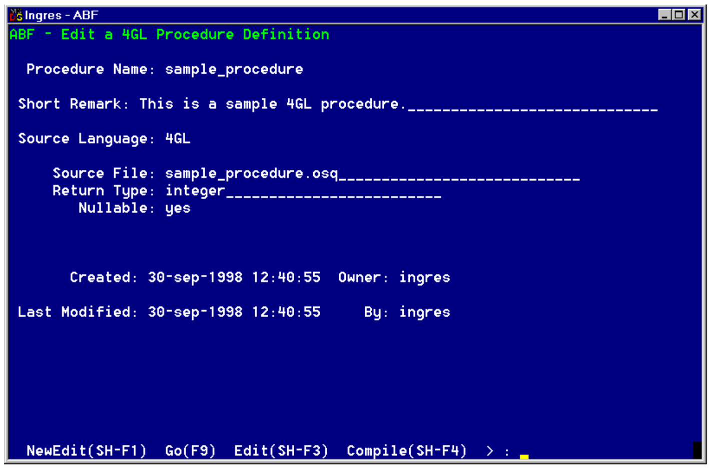
1. From the Edit an Application frame, choose Create. The pop-up Create a Frame or Procedure frame appears.
2. Highlight Procedure and choose Select. This displays a pop-up frame with the types of available procedures, as shown in the Create a Procedure pop-up figure.
3. Select the type of procedure by highlighting the correct type and choosing Select. If you select the OtherLanguages menu operation from the menu line, the pop-up frame displays a list of all callable non-embedded languages supported by your site, from which you can select.

4. The Create a *Proceduretype* Procedure pop-up appears for the type of procedure you selected. The Create a 4GL Procedure frame is shown in the Create a 4GL Procedure pop-up figure.
5. In the Name field, enter the name of this procedure. The other fields are display only.
6. Choose OK. ABF creates the procedure and displays the Edit a Procedure Definition frame for this procedure type.

Important! Avoid giving a procedure the same name as a system function or built-in frame or procedure, as this overrides the system component. A pop-up message warns you if you use a system name.

Edit a Procedure Definition Frame

After you create a procedure using the Create a *Proceduretype* Procedure frame, you see the Edit a *Proceduretype* Procedure Definition frame for the type of procedure you have chosen. The following figure shows the Edit a Procedure frame for the type 4GL:



```
ABF - Edit a 4GL Procedure Definition

  Procedure Name: sample_procedure

 Short Remark: This is a sample 4GL procedure._____________________________

 Source Language: 4GL

     Source File: sample_procedure.osq_____________________________
     Return Type: integer_________________________
        Nullable: yes


       Created: 30-sep-1998 12:40:55  Owner: ingres

 Last Modified: 30-sep-1998 12:40:55     By: ingres




  NewEdit(SH-F1)  Go(F9)  Edit(SH-F3)  Compile(SH-F4)  > :
```

The Edit a *Proceduretype* Procedure frames have similar fields. The following list shows the fields as they appear on the frame (some fields do not appear for all procedure types, as indicated in the list).

Procedure Name

Specifies the name of procedure you entered in the Create a Procedure frame

Short Remark

(Optional) Specifies a brief description

Source Language

Specifies either 4GL, SQL (for database procedures), or the name of a 3GL supported in your installation

Symbol field

(For 3GL procedures only) Specifies the link symbol ABF uses to link to the procedure. The default is the procedure name. The name is case sensitive on systems whose linkers are case sensitive.

UNIX: On some UNIX systems, you must add an underscore (_) after the symbol name. Check your operating system documentation for more information on specifying symbol names.

Library field

(For 3GL procedures only) Indicates whether this is a library procedure. A library procedure is not compiled by ABF, but must be defined to ABF. If you enter yes, the Edit and Compile menu operations are not active.

Source File field

(For 3GL and 4GL procedures only) Specifies the file name with the extension of the selected language, as shown in the table "File Extensions". This field is case sensitive and can contain up to 48 characters. Do not enter a value in this field for library procedures, including database procedures.

Return Type

Specifies the default value is integer. You can enter none, integer, float or any Ingres string type. The decimal data type is supported for 4GL procedures only.

Nullable

Determines whether the field can hold a null value. The Nullable field is display only with a value of "No" for all 3GL procedures, because these do not support nullable return values.

Pass Decimal Values As field

(For COBOL procedures only) Specifies whether the procedure passes return values with decimal data types as float data type or decimal data type.

All procedure types have the Created and Last Modified fields. The Last Modified field indicates when the procedure's definition frame was last modified.

The Edit a *Proceduretype* Procedure frames have the following menu operations:

NewEdit

Edits or creates the definition of a different component. Prompts for the name and type of component to define.

Go

Runs the procedure. For 3GL procedures, you cannot use Go if the procedure receives parameters from the calling frame. ABF warns you that attempting to run a procedure that expects a parameter can have unpredictable results. To test a 3GL procedure that requires parameters, run the procedure from the calling frame.

Edit

Uses the system editor to create or edit the file containing the source code for the current procedure. (Does not appear for database procedures.)

Compile

Checks the source code syntax for the current procedure, displays errors (if any), and adds the object code for the procedure to the database if free of errors. (Does not appear for database procedures.)

LongRemark

Displays or enters more information about the current procedure on the LongRemark pop-up frame

Print

Prints the definition and source code file for the procedure

ListChoices, Cancel, Help, End

Perform standard operations

Define a 4GL Procedure

To use the Edit a 4GL Procedure Definition frame to define a procedure

1. The Procedure Name field contains the procedure name you entered in the Create Frame.
2. (Optional) Enter a Short Remark.
3. In the Source File field, either leave the default file name, made up of the procedure name and the extension .osq, or enter another file name.
4. The Return Type field contains the default value integer. Enter the data type of data you intend to return from this procedure to the calling frame or procedure, or none if no data is to be returned.

   To select the return type from a list, choose ListChoices.

5. Enter Yes or No in the Nullable field to indicate whether or not NULL can be returned. The default value is "Yes."
6. Optionally choose LongRemark to enter more detailed information about the procedure. Select Save and End to return to the Edit a 4GL Procedure Definition frame.

Enter Source Code

To enter the source code for a 4GL procedure

1. Choose Edit from the Edit a 4GL Procedure Definition frame menu to call up the system editor's opening display.
2. Enter the source code for the procedure.
3. When you finish, save your file as you exit the editor.
4. You return to the Edit a 4GL Procedure Definition frame. Choose Compile to compile the code and check for syntax errors.

   If the file contains compilation errors, ABF displays a frame that lists the errors and prompts you to correct them. It is the same frame that you see if errors occur during creation of a user frame. Enter **y** (yes) to see the 4GL Error Listing frame.
5. Test the procedure by running the frame that calls it.

Create 3GL Procedures

The Create a *languagename* Procedure frame has similar fields for all 3GLs. The title of the frame specifies the language you are using to create the procedure. The language is a supported programming language, such as Ada, C, COBOL and Fortran.

An Ada procedure must be a library procedure.

The only difference between the fields on the Create a *languagename* Procedure frame for different languages is that the Pass Decimal Values As field only appears for COBOL procedures.

To define a 3GL host language procedure

1. The Procedure Name field contains the name you entered in the Create a Procedure frame.
2. (Optional) Enter a brief description in the Short Remark field.
3. In the Source Language field, enter the supported language in which you are writing this procedure.
4. In the Symbol field, enter the link symbol for ABF to use to link to the procedure.

5. In the Library field, enter **yes** or **no** to indicate whether this is a library procedure or not. If you enter yes, the Edit and Compile menu operations are not active.
6. In the Source File field, enter a source code file name with the extension of the selected language, as shown in the table "File Extensions".
7. For COBOL only: enter decimal or float in the Pass Decimal Values As field.
8. Choose Edit to open a file in the system editor. Enter the source code.
9. To select the Return Type from a pop-up list, choose ListChoices. Highlight the desired type, enter a length if required for the data type, and enter Select.

   The Nullable field is display only.
10. Optionally choose LongRemark to display or add detailed information about the procedure on the LongRemark pop-up.
11. To test the procedure, choose Compile as described above for 4GL procedures.

Using Embedded SQL/Forms with ABF

You can perform most forms control through 4GL. However, if you must use forms control which is not supported in 4GL, you can call an ESQL procedure which displays a form. The table "File Extensions" shows the languages in which the forms-handling facilities of SQL are available.

If the ESQL procedure uses a form that is not used by any user-specified frame, it must initialize the form with exec sql forminit or exec sql addform. In addition, a compiled form that is not used by any user-specified frame must be specified in a link-options file. For more information, see ABF Architecture (see page 501).

An ESQL procedure can operate on any form that has been displayed during the current session, whether the frame is currently visible or not.

The ESQL procedure must use an exec sql display statement to display the form. The ESQL code in ESQL procedures cannot perform exec sql connect, exec sql disconnect, exec frs forms or exec frs endforms statements. For details, see the *SQL Reference Guide*.

Creating a Database Procedure

The procedure for creating database procedures and 4GL procedures are similar. Highlight SQL on the Create a *Proceduretype* Procedure frame to display the Edit a Database Procedure Definition frame.

You create the database procedure outside of ABF, using the SQL create procedure statement. See the *SQL Reference Guide* for more information on creating database procedures.

The Edit a Database Procedure Definition frame does not contain the Edit or Compile operations. The remaining operations are described above for the other procedure types.

To complete the Edit a Database Procedure Definition frame, follow the steps in Define a 3GL Procedure (see page 171) that apply to Database procedure fields. Choose Create to create an entry for the procedure in the ABF system catalogs.

The Globals Submenu of the Edit an Application Frame

The Globals operation of the Edit an Application Frame menu displays the Globals menu on the menu line. Through this menu, you can define and edit global variables, constants, and record types for your applications. See the Introduction chapter for a discussion of these terms.

When planning your application, consider where you can use these global components.

Global variables

If your application makes repeated use of a certain value—for example, the name of the current user of the application—you can define this value as a global variable. After you create a global variable, you must write 4GL code (usually in an application's starting frame or procedure) to set its initial value. You then can change the value of the global variable while the application is running.

A global variable can have as its data type a simple
Ingres type (for example, integer or character), a record type that you have created, or an array.

Global variables can be used wherever local variables are used. However, you must use a local variable unless the variable is used in several frames. Replacing all local variables with global variables increases compilation time.

Constants

If your application uses the same value several times—for example, the current ales tax rate—you can store this value in an application constant. You set the value of a constant when you create it; this value cannot be changed while the application is running.

Global constants must have a simple Ingres data type. They cannot be date or money data types or of a record type.

Record types

If certain grouping of related information occurs together throughout an application, you can group these in a record type. You can treat the objects in a record type as a single object, creating, for example, a combination of Business Name and City, or Customer Name, Address, and City.

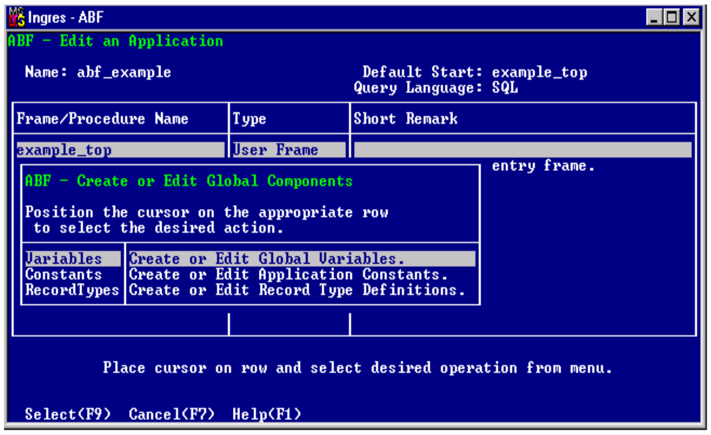
In naming these components, consider these issues:

- A global variable cannot have the same name as a global constant. Local variable names override global variable and constant names when these conflict.
- Record types can have names that conflict with those of other components.

You can use global components in any frame or procedure of an application. For information on using 4GL code to reference and manipulate these global components, see the 4GL section of this guide.

The Create or Edit Global Components Frame

To define global variables, constants, and record types for your application, select the Globals option from the Edit an Application Frame menu. You see the Create or Edit Global Components frame, shown in the following figure:



```
Ingres - ABF
ABF - Edit an Application

  Name: abf_example                        Default Start: example_top
                                          Query Language: SQL

 Frame/Procedure Name | Type       | Short Remark
 example_top          | User Frame |
  ABF - Create or Edit Global Components          entry frame.

  Position the cursor on the appropriate row
   to select the desired action.

  Variables   Create or Edit Global Variables.
  Constants   Create or Edit Application Constants.
  RecordTypes Create or Edit Record Type Definitions.

           Place cursor on row and select desired operation from menu.

  Select(F9)  Cancel(F7)  Help(F1)
```

The following table shows the options available from the Globals submenu:

Variables

Creates or edits global variable declarations for the application

Constants

Creates or edits constant declarations for the application

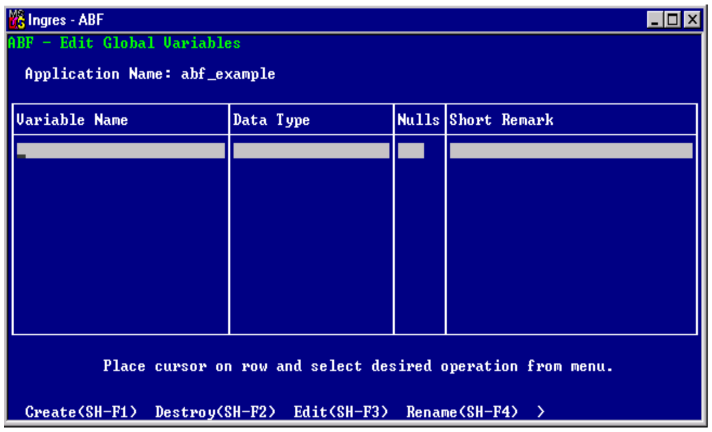
RecordTypes

Creates or edits record type definitions for the application

The menu line of the underlying frame shows the standard operations, Select, Cancel, and Help.

The Edit Global Variables Frame

When you select Variables from the Create or Edit Global Components Frame, you see the Edit Global Variables frame, shown in the following figure. From this frame, you can manage the global variables you define and use in an application.



```
ABF - Edit Global Variables

  Application Name: abf_example

 Variable Name            Data Type          Nulls Short Remark

          Place cursor on row and select desired operation from menu.

  Create(SH-F1)  Destroy(SH-F2)  Edit(SH-F3)  Rename(SH-F4)  >
```

All the fields on this form are read-only. The global variables are displayed in the table field in alphabetical order, with the following information:

Variable Name

Specifies the name of the global variable. This column is scrollable.

Data Type

Specifies the data type of the global variable. This column is scrollable.

Nulls

Indicates whether the global variable is nullable

Short Remark

(Optional) Provides a description of the variable. This column is scrollable.

The Edit Global Variables frame offers the following menu operations:

Create

Creates a new global variable. Displays the Create a Global Variable frame.

Destroy

Destroys the selected global variable. Prompts you to confirm the decision.

Edit

Edits the selected global component. Calls a frame that allows you to edit the selected variable.

Rename

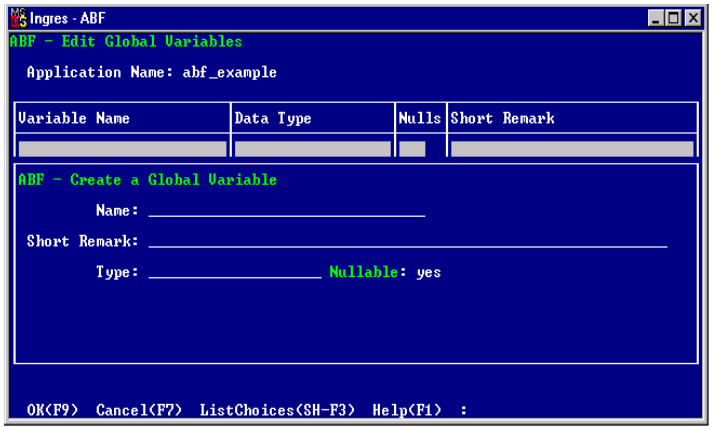
Renames the selected global variable. Prompts you for a new variable name. Press Return to cancel the Rename.

Help, End

Perform standard operations

The Create a Global Variable Frame

To create a global variable for the application, select the Create menu item from the Edit Global Variables frame menu. The Create a Global Variable pop-up frame appears below. Enter the name, type, and description of the global variable.



```
ABF - Edit Global Variables

  Application Name: abf_example

 Variable Name           Data Type          Nulls Short Remark

 ABF - Create a Global Variable

          Name: ________________________________

  Short Remark: ____________________________________________________________

          Type: ____________________ Nullable: yes

  OK(F9)  Cancel(F7)  ListChoices(SH-F3)  Help(F1)  :
```

The Create a Global Variable frame has the following fields:

Name

(Required) Specifies the variable name. This field has the following restrictions. The first two are validated on the form; the last is verified when you select OK. The name:

- Must follow standard object naming conventions.
- Cannot begin with "ii."
- Cannot be the name of another *global variable* or *constant, frame,* or *procedure* defined for the application.

Short Remark

(Optional) Provides a brief description. The Short Remark appears on the Edit Global Variables catalog frame.

Type

(Required) Specifies the name of any record type defined for this application or any Ingres type. When you first specify an attribute, this field is empty. The data type you enter here determines whether you fill in the Nullable or the Array field next.

- If you enter an Ingres data type, you see the Nullable field.
- If you enter the name of a record type created for the application, you see the Array field.

Nullable

Indicates whether the global variable is nullable. Only variables of an Ingres type can also be nullable. Record types cannot be nullable. The default depends on the application query language: "yes" for SQL and "no" for QUEL.

Array

Indicates whether this global variable is a single record or a dynamic array of records. Arrays cannot be composed of Ingres data types.

The following menu operations are available from this frame:

OK

Creates the new global variable

Cancel

Cancels the creation and return to the previous frame without saving changes

ListChoices

Displays a pop-up menu of legal values for the field or a field description. If prompted (because you have selected a character data type), enter the size in characters on the ListChoices pop-up before choosing OK to enter the data type in the field.

Help

Displays help for this frame

When you have successfully created the new global variable, ABF inserts its description into the Edit Global Variables frame catalog table field and then displays the Edit a Global Variable Definition frame.

Create a Global Variable

To create a global component, select the Create menu item from the Edit Global Variables Frame. You see the Create a Global Variable frame.

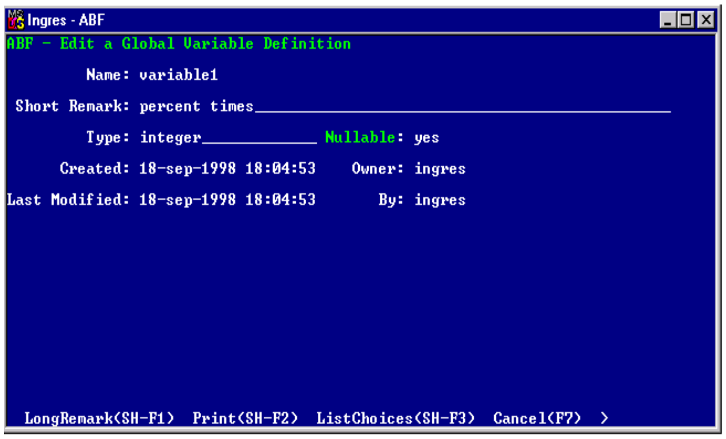
To create a global variable

1. In the required Name field, enter a name that meets the restrictions noted in the previous section.
2. In the required Type field, enter the name of any record type defined for this application or any legal Ingres type. To select the data type from a list, choose ListChoices.
3. You either see the Array or the Nullable field, depending on the data type you enter. For more information, see The Create a Global Variable Frame (see page 397).
4. Select OK to complete the creation of the global variable. All field values must be valid before creation can proceed. This includes ensuring that the Name is unique. You return to the Edit Global Variables catalog frame, where the new global variable appears in the table field.

You are now ready to edit the global variable you have created.

Edit a Global Variable Definition Frame

From the Edit Global Variables catalog frame, you can modify the definition of a global variable by highlighting the name of the variable and selecting the Edit menu item. You see the Edit a Global Variable Definition frame, shown in the following figure:



```
ABF - Edit a Global Variable Definition

         Name: variable1

 Short Remark: percent times_______________________________________________

         Type: integer_____________ Nullable: yes

      Created: 18-sep-1998 18:04:53    Owner: ingres

Last Modified: 18-sep-1998 18:04:53       By: ingres
```

Ingres - ABF
LongRemark(SH-F1) Print(SH-F2) ListChoices(SH-F3) Cancel(F7) >

This frame contains the following fields:

Name

(Read-only) Displays name of the global variable you entered on the Create a Global Variable frame.

Short Remark

(Optional) Provides a brief description

Type

(Required) Specifies the data type of the global variable. The name of any record type defined for this application or any Ingres type. ABF checks this whenever the value is changed on the form. The data type you enter here determines whether you fill in the Nullable or the Array field next.

- If you enter an Ingres data type, you see the Nullable field.
- If you enter the name of a record type created for the application, you see the Array field.

Nullable

Indicates whether the global variable is nullable. Only attributes of an Ingres type can also be nullable. Record types cannot be nullable. The default depends on the application query language: "yes" for SQL and "no" for QUEL.

Array

Indicates whether this global variable is a single record or a dynamic array of records. Arrays cannot be composed of Ingres data types.

The following menu operations are available from this frame:

LongRemark

Reads or edits the Long Remark pop-up frame

Print

Prints the definition of the global variable

ListChoices

Displays a list of legal values for the field from which you can choose *or* a field description

Cancel, Help, End

Perform standard operations

Edit a Global Variable Definition

You can change the features of an existing global variable with the Edit a Global Variable Definition frame.

To edit a global variable definition

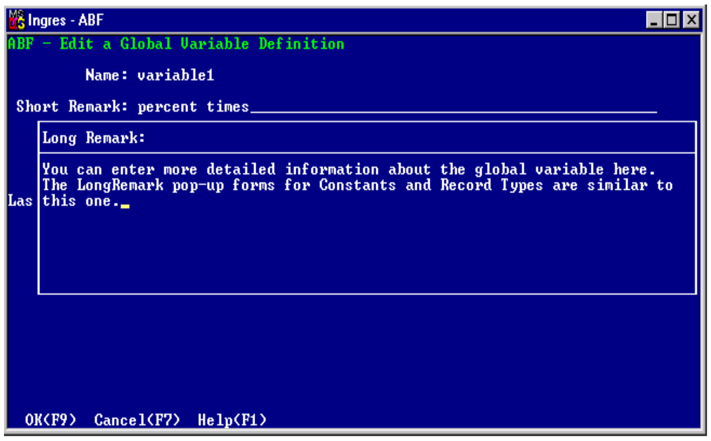
1. Select the Edit menu item from the Edit Global Variables frame to display the Edit a Global Variable frame.

   The Name field is display only. You cannot change this here.

2. In the required Data Type field, enter the name of any Ingres data type or any record type defined for this application.

3. The next field is either the Nullable or the Array field, depending on the data type you enter. For more information, see The Edit a Global Variable Definition Frame section.

4. You can modify the optional Short Remark here.
5. You can enter or view more detailed information about the global variable by selecting the LongRemark menu operation. This displays the LongRemark pop-up, shown in the following figure:



```
Ingres - ABF
ABF - Edit a Global Variable Definition

        Name: variable1

 Short Remark: percent times_______________________________________________

    Long Remark:

    You can enter more detailed information about the global variable here.
    The LongRemark pop-up forms for Constants and Record Types are similar to
Las this one._

  OK(F9)  Cancel(F7)  Help(F1)
```

Constants and Record Types have similar LongRemark pop-ups.

How to Set Initial Values for Global Variables

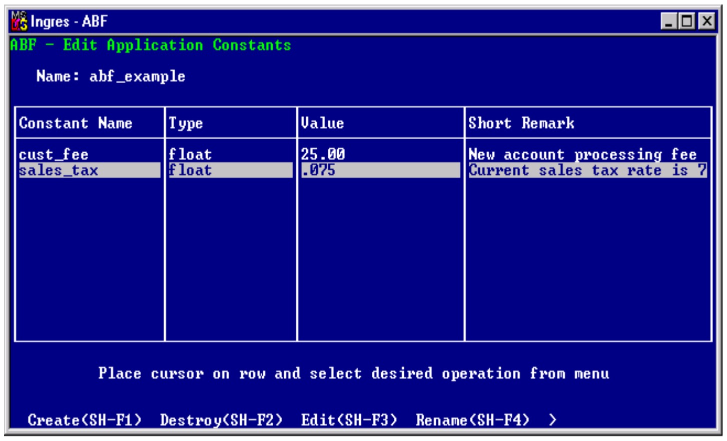
After you create a global variable, you must set its initial value before using it in your application. You must do this in the 4GL code for the application's starting frame or procedure. This ensures that you have initialized it before the application calls any frames or procedures that use the global variable.

When testing a frame that uses a global variable, start the application with the frame that initializes it. Then, proceed to the frame that you want to test. Otherwise, the test frame cannot run properly, because the statements that initialize its global variables have not been run.

See the 4GL section of this guide for a procedure that you can use to check whether you have set the initial value of a global variable. You can call this procedure from any frame or procedure that calls the global variable.

The Edit Application Constants Frame

To define constants for your application, select the Constants menu item from the Globals submenu. ABF calls the Edit Application Constants frame. Constants defined using this frame, shown below, are global to the application. See the 4GL section of this guide for information on using global constants in your application's 4GL code.



```
Ingres - ABF
ABF - Edit Application Constants

   Name: abf_example

Constant Name    Type            Value                Short Remark
cust_fee         float           25.00                New account processing fee
sales_tax        float           .075                 Current sales tax rate is 7

          Place cursor on row and select desired operation from menu

  Create(SH-F1)  Destroy(SH-F2)  Edit(SH-F3)  Rename(SH-F4)  >
```

The catalog table field lists each constant defined for the application in alphabetical order by name. All the fields on this form are display only.

Name

Displays the application name

Constant Name

Displays the name of the constant. This column is scrollable.

Type

Displays the data type of the constant

Value

Displays the value of the constant. For example, if the constant represents a menu item, the value is the string that appears in the window for the menu item. The Value column is scrollable. If you want, you can have alternate sets of values for each constant. See Using Alternate Sets of Global Constant Values (see page 407).

Short Remark

(Optional) Displays a brief description, a scrollable field

The following menu operations are available from this frame:

Create

Creates a new constant. Displays the Create an Application Constant frame.

Destroy

Destroys the selected constant. Prompts you to confirm the decision.

Edit

Edits the selected constant. Calls the Edit a Constant Definition frame.

Rename

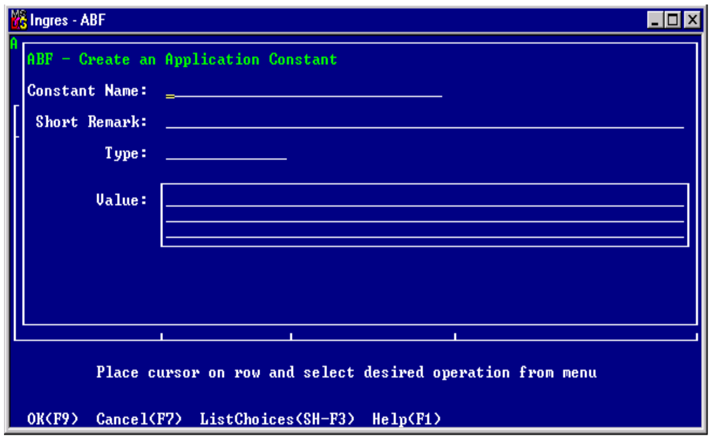
Renames the selected constant. Prompts you for a new name. Press Return to cancel the Rename.

Help, End

Perform standard operations

The Create an Application Constant Frame

To create an application constant, select Create from the Edit Application Constants frame. The Create an Application Constant pop-up frame appears, as shown in the following figure. This frame is identical to the Create a Global Variable frame, except that this frame has an additional Value field.



```
ABF - Create an Application Constant

Constant Name: _
Short Remark:
Type:

Value:

Place cursor on row and select desired operation from menu

OK(F9)  Cancel(F7)  ListChoices(SH-F3)  Help(F1)
```

This frame has the following fields:

Constant Name

(Required) Specifies a name that follows the standard naming conventions and is not the name of another constant, global variable, frame, record type, or procedure. The first requirements are validated on the form; the other is verified when you select OK.

Short Remark

(Optional) Provides a brief description

Type

(Required) Specifies the data type of the constant. Valid types for constants are decimal, integer, float, char, varchar, and smallint. ABF does not support Null constants. The field is blank when you enter the frame.

Value

(Required) Specifies a constant value that agrees with the type in the Type field. For example, "xyz" is not a valid value for an integer constant. The field is blank when you enter the frame. The maximum constant value is displayed underlined on three lines.

This frame has the following available menu operations:

OK

Creates the new constant

Cancel

Cancels the creation and return to the previous frame

ListChoices

Displays a list of legal values for the field from which you can choose *or* field data specifications

Help

Displays help for this frame

Create an Application Constant

To create an application constant, select the Create menu item from the Edit Application Constants frame to display the Create an Application Constant pop-up frame.

To create an application constant

1. In the required Name field, enter a name that meets the limitations described in the previous section. The name must be unique.
2. You can enter a brief description of the constant in the Short Remark field.

3. In the required Type field, enter the data type of the constant.

   If the type requires a length, indicate the length in parenthesis. If you don't specify a length, ABF uses the length of the value you specify in Step 4.

4. In the required Value field, enter a constant value that agrees with the type in the Type field.

   If you specify a value longer than the length you specified in the Type field, ABF displays a pop-up asking if you want to truncate the value or increase the length of the data type.

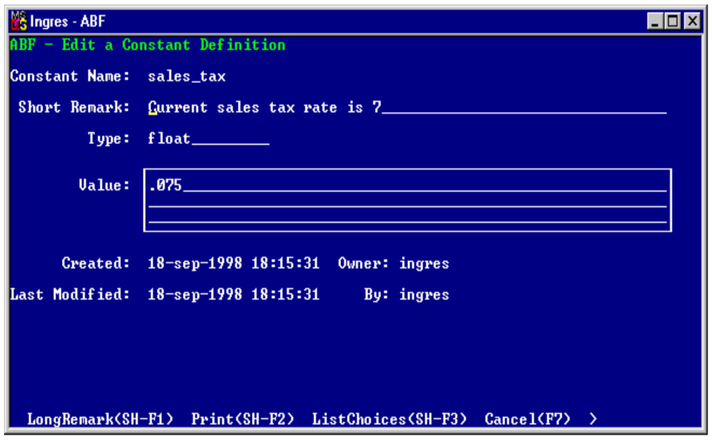
5. Make sure all field values are valid before you select OK. Otherwise, ABF does not create the constant.

6. Select OK to create the constant. Cancel, instead of OK, terminates the create operation and returns to the Edit Application Constants frame.

Following successful creation, ABF displays the Edit Application Constants frame table field, where the description of the new constant is listed.

The Edit a Constant Definition Frame

Once you create a constant, you can define and modify it by selecting the Edit menu item in the Edit Application Constants frame, shown in the Edit Application Constants frame figure. ABF calls the Edit a Constant Definition frame, shown in the following figure, for the constant you select.


```
ABF - Edit a Constant Definition

Constant Name:  sales_tax

 Short Remark:  Current sales tax rate is 7_____________________________

         Type:  float_________

        Value:  .075____________________________________________________

      Created:  18-sep-1998 18:15:31   Owner: ingres

Last Modified:  18-sep-1998 18:15:31      By: ingres

  LongRemark(SH-F1)  Print(SH-F2)  ListChoices(SH-F3)  Cancel(F7)  >
```

The values for the fields on this frame are supplied by the Create an Application Constant frame.

Name

(Read-only) Displays the name of the constant

Short Remark

(Optional) Provides a brief description

Type

(Read-only) Displays the data type of the constant; any Ingres type

Value

Specifies the value that the constant represents

The following menu operations are available from this frame:

LongRemark

Displays or edits more information about the constant on the Long Remark pop-up form

Print

Prints the definition of the global variable

ListChoices

Displays a list of legal values for the field from which you can choose *or* field data specifications

Cancel, Help, End

Perform standard operations

Using Alternate Sets of Global Constant Values

Your application typically uses the default set of global constant values defined in the application. You can set up alternate values for the global constants. For example, you can define and use constants for all strings you use in an application. You can create one set for each language you want to use.

The default set of constant values is stored in the database. You can create alternate sets of constant values and store them in files. You can override the default constants by specifying the file name before running an application or creating an image of an application.

When a global constant file is specified before running the application or creating an image, the values in the constant file override the default values stored in the database. The values in the constants file are in effect until you clear the field, specify a different filename, or edit a different application.

Specify a Global Constants File

You can specify a global constants file when you use the imageapp command or when you run the application image. You can also specify a global constants file within ABF, before choosing Go or Image from the menu.

To specify a global constants file within ABF

1. Select Utilities from the menu in the Edit an Application window or the Applications Catalog window.
2. Select ConstantsFile from the menu.
3. Specify a filename in the Global Constants File Name field.
4. Select End.
5. To run the application with the specified constants file:
   a. Select End.
   b. Select Go.

To create an image with the specified constants file, select Image.

Create an Alternate Global Constants File

You can create a global constants file with your system editor, or copy the default global constant values (that you defined through the Edit Global Constants window) into a file.

To create an alternate set of global constants from the default set

1. Select Utilities from the menu in the Applications Catalog window or Application Flow Diagram Editor.
2. Select ConstantsFile from the menu.
3. Specify a filename in the Global Constants File Name field.
4. Select Create to copy the default global constants into a file.

To create an alternate set of global constants with your system editor

Create and edit a file with the following format:

- A constant name, which must begin in column 1.
- The character : or =. White space can appear on either side of this character.
- A value for the constant. The value can continue across multiple lines, but continuation lines must begin with a space, or tab.

For example:

```
menu1: 'This is the first menu item'
menu2 = 'Menu2 is the second menu item'
```

The file can also contain comments beginning with a # character. For more information about the global constants file format, see the *SQL Reference Guide* section on the PM file format.

Edit an Alternate Set of Global Constants

You change the alternate set of global constant values by editing the constants file. You cannot edit the alternate set of values through the Edit Global Constants window. You can edit the file with your system editor, from your operating system or from within Vision.

To edit the file from within Vision

1. Select Utilities from the menu in the Applications Catalog window or Application Flow Diagram Editor.
2. Select ConstantsFile from the menu.
3. Specify a filename in the Global Constants File Name field.
4. Select Edit from the menu to edit the specified file.

When you edit or destroy a constant in one file, it does not affect the constants in another file, or the default values in the database.

Compatibility with Previous ABF Releases

Prior to the current release of ABF, specify alternate languages in the Edit Global Constants window. Multiple sets of language-dependent constants values were stored in the database. In this release, only one set of global constants values are stored in the database, and alternate sets are stored in separate files. If you have a previous release of the database that contains multiple language values, the values are retained in the catalogs.

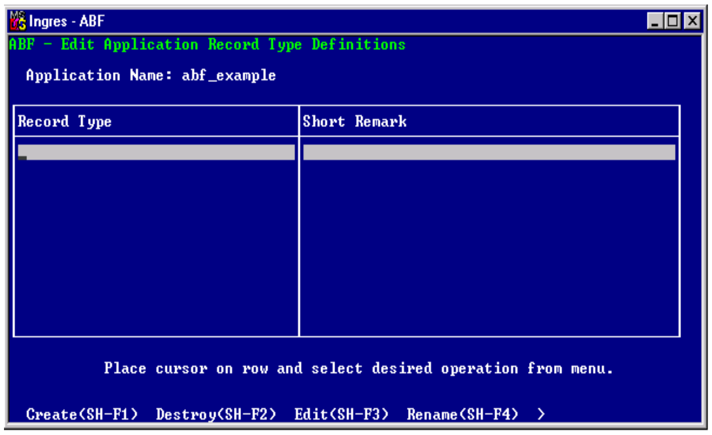
To convert global constants to the new format, create a file of global constants from the default set, as described in Create an Alternate Global Constants File (see page 408). ABF creates a file with the global constants values separated by language, as shown in the example below. The data type is provided as a comment. You can then copy each set of values into its own global constants file.

```
# PM readable Global Constant File
#
# Application name: ccf
#
```

```
#BEGIN  default
 cons1                    ='English'         #char(7)
 cons2                    ='yards '          #char(6)
 cons3                    ='Fahr'            #char(4)
 #END   default


#BEGIN French
 cons1                    ='French '         #char(7)
 cons2                    ='meters'          #char(6)
 cons3                    ='Cels'            #char(4)
 #END French


#BEGIN german
 cons1                    ='German '         #char(7)
 cons2                    ='meters'          #char(6)
 cons3                    ='Cels'            #char(4)
#END german

# Number of Global Constants: 3
```

The Edit Application Record Type Definitions Frame

When you select RecordType from the Globals submenu, you see the Edit Application Record Type Definitions frame, shown below. Use this frame to manage the record types you define for an application. The table field lists the names of all the record types defined for the application with a Short Remark.



```
Ingres - ABF
ABF - Edit Application Record Type Definitions

  Application Name: abf_example

 Record Type                          Short Remark


              Place cursor on row and select desired operation from menu.

  Create(SH-F1)  Destroy(SH-F2)  Edit(SH-F3)  Rename(SH-F4)  >
```

All the fields on this form are read-only. The rows are displayed in the table fields in alphabetical order by record type name.

Record Type

Displays the names of the record types defined for the application

Short Remark

(Optional) Provides a brief description of the record type. This column is scrollable.

The following menu operations are available from this frame:

Create

Creates the definition of a new record type. Displays the Create a Record Definition pop-up frame for the record type.

Destroy

Destroys the selected record type. Prompts you to confirm the decision.

Edit

Edits the selected record type. Displays an attribute catalog frame for the selected record.

Rename

Renames the selected record type. Prompts you for a new record type name. Press Return to cancel the Rename.

MoreInfo

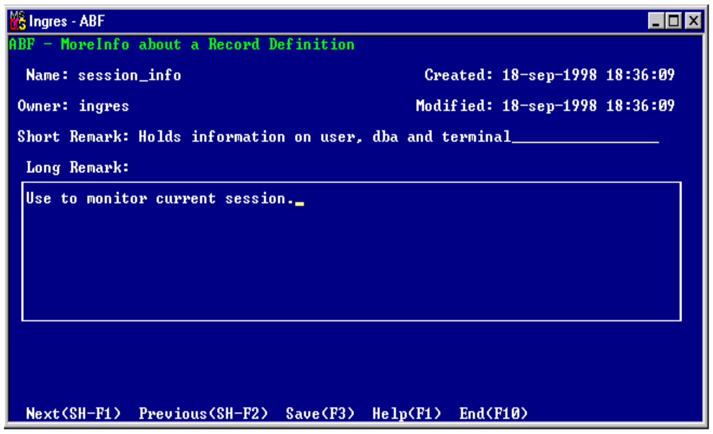
Displays or edits more information about the record type on the MoreInfo About a RecordType frame.

Help, End

Perform standard operations

MoreInfo About a Record Definition Frame

You can display or enter more information about any record type that appears in the Edit Application Record Type Definitions frame. To see this frame, highlight the record type name, enter the menu line, and select the MoreInfo operation. An example appears in the following figure:



```
ABF - MoreInfo about a Record Definition

 Name: session_info                               Created: 18-sep-1998 18:36:09
Owner: ingres                                    Modified: 18-sep-1998 18:36:09
Short Remark: Holds information on user, dba and terminal_________________
 Long Remark:
 Use to monitor current session._

 Next(SH-F1)  Previous(SH-F2)  Save(F3)  Help(F1)  End(F10)
```

The following information appears in *display-only* form. You use other ABF frames to enter the information on this frame.

Name

Displays the name of the record type

Owner

Displays the the application's owner

Created

Displays the date the record type was created

Modified

Displays the date the record type was last modified

You can enter and modify the following fields:

Short Remark

Specifies a one-line description of the record type that appears in the Edit Application Record Type Definition frame table field

Long Remark

Specifies a comment of up to eight lines

The frame offers the following menu operations:

Next

Displays details on the next record type listed on the Edit Application Record Type Definition frame table field. ABF prompts you to save changes before moving.

Previous

Displays details on the previous record type listed on the Edit Application Record Type Definition frame table field. ABF prompts you to save changes before moving.

Save

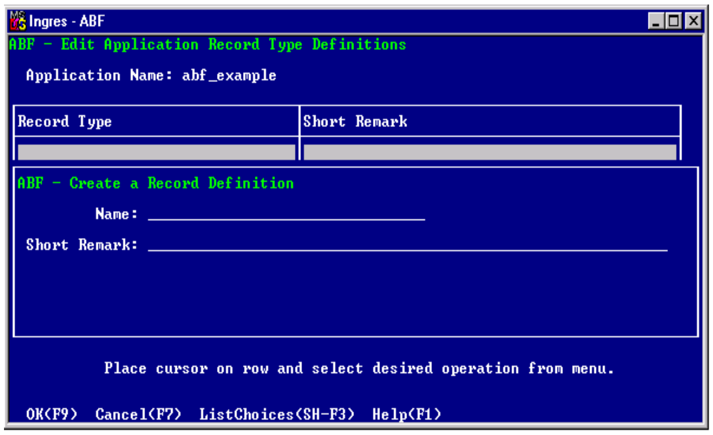
Saves changes made on this frame

Help, End

Perform standard operations

The Create a Record Definition Frame

To define a record type for your application, select Create from the Edit Application Record Type Definitions frame. Create displays the Create a Record Definition pop-up frame, on which you can enter the name and description. See the following figure:



```
Ingres - ABF
ABF - Edit Application Record Type Definitions

  Application Name: abf_example

 Record Type                        Short Remark

 ABF - Create a Record Definition

          Name: ________________________________

  Short Remark: ____________________________________________________________

          Place cursor on row and select desired operation from menu.

  OK(F9)  Cancel(F7)  ListChoices(SH-F3)  Help(F1)
```

This pop-up has only two fields:

Name

(Required) Specifies a name that follows the standard naming conventions and is not the name of a Ingres data type or an existing record type for the application. The field is blank when you first see it.

Short Remark

(Optional) Provides a brief description of the record type. This remark appears in the Edit Application Record Type Definitions table field, and can be modified through the MoreInfo About a Record Definition frame.

The menu operations are identical to those on the Create an Application Constant frame:

OK

Creates the new record type

Cancel

Cancels the creation and return to the previous frame

ListChoices

Displays a field data description

Help

Accesses the Help facility

Create Record Types

Start by selecting the Globals operation from the Edit an Application frame. Select RecordType from the resulting submenu.

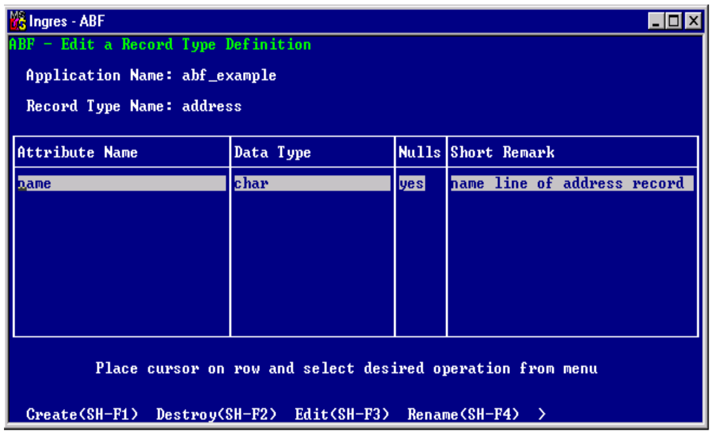
To define a record type

1. From the Edit Application Record Type Definitions frame, shown in Edit Application Record Type Definitions frame figure, select Create. The Create a Record Definition frame appears, as shown in the previous figure.
2. In the Name field, enter a name. This cannot be the name of an Ingres data type or a record type for the application. If the record Name is not unique, you must create the record definition again. The field is blank when you first see it.
3. You can enter and modify the Short Remark field on this frame.
4. Create the record by selecting OK. To stop the operation, select Cancel, which returns you to the Edit Application Record Type Definitions frame.

Following the successful definition of a record type, ABF displays the Edit Application Record Type Definitions catalog frame. You can use this frame to define the attributes for the record type.

The Edit a Record Type Definition Frame

After you have created or selected a record type for editing, the Edit a Record Type Definition frame appears. Use this catalog frame to add, delete, or modify the attributes of a record type. The attributes are displayed in a table field, along with their data type, nullability, and short remarks. See the following figure:



```
ABF - Edit a Record Type Definition

  Application Name: abf_example

  Record Type Name: address

 Attribute Name           Data Type          Nulls Short Remark
 name                     char               yes   name line of address record

          Place cursor on row and select desired operation from menu

  Create(SH-F1)  Destroy(SH-F2)  Edit(SH-F3)  Rename(SH-F4)  >
```

The fields in this frame are read-only fields. The table field displays the attributes defined for the record type alphabetically by attribute name.

Application Name

Specifies the name of the application

Record Type Name

Specifies the record to which the attributes in this catalog frame belong

Attribute Name

Specifies the names of the attributes. This column is scrollable.

Data Type

Specifies the data types of each attribute, a scrollable column

Nulls

Indicates the nullability of each attribute

Short Remark

Displays a brief description of each attribute, a scrollable column.

This frame provides the following menu operations:

Create

Creates the definition of a record attribute. Displays the Create a Record Attribute frame for the record type.

Destroy

Destroys the selected record attribute. Prompts you to confirm the decision.

Edit

Edits the selected record attribute. Calls the Edit a Record Attribute Specification frame for the existing attribute in the highlighted row.

Rename

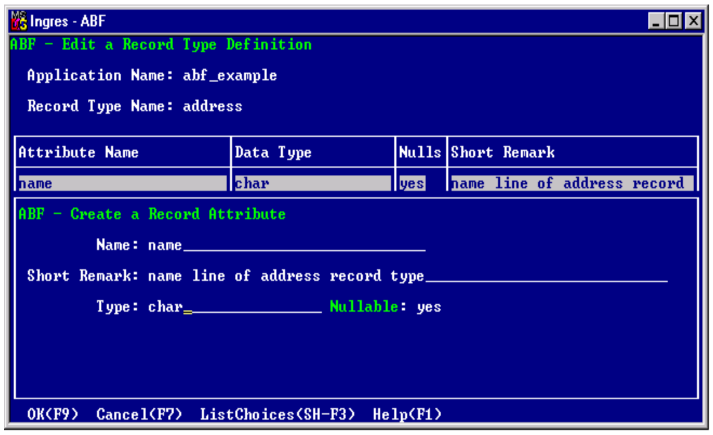
Renames the selected record attribute. Prompts you for an attribute name. Press Return to cancel the Rename. You cannot give the attribute the name of an Ingres data type.

Help, End

Perform standard operations

The Create a Record Attribute Frame

To specify a record attribute, use the Create a Record Attribute frame. Display this frame from the Edit a Record Attribute frame by selecting the Create menu operation. The Create a Record Attribute frame is shown in the following figure:



```
ABF - Edit a Record Type Definition

  Application Name: abf_example

  Record Type Name: address
```

Attribute Name	Data Type	Nulls	Short Remark
name	char	yes	name line of address record

```
ABF - Create a Record Attribute

         Name: name

 Short Remark: name line of address record type

         Type: char            Nullable: yes
```

ABF does not allow a record type to contain an attribute of the same record type as itself, nor can its attributes be records that include an attribute of the parent type. For example, if record type A includes an attribute of type C, C cannot contain an attribute of record type A.

The scope of a record attribute is the record itself. No two attributes for a record can have the same name; however, attributes in different records can have the same name. An attribute type can be:

- Any Ingres data type.
- Any other record type defined for the application.
- An array of any other record type.

This frame has the following fields:

Name

(Required) Specifies the name of the record attribute

Short Remark

(Optional) Provides a brief description

Type

(Required) Specifies the name of any record attribute defined for this application or any Ingres type. The data type you enter here determines whether you fill in the Nullable or the Array field next.

- If you enter an Ingres data type, you see the Nullable field.
- If you enter the name of a record type created for the application, you see the Array field.

Nullable

Indicates whether the record attribute is nullable. Only attributes of an Ingres type can also be nullable. Record types cannot be nullable. The default depends on the application query language: "yes" for SQL and "no" for QUEL.

Array

Indicates whether this attribute is a single record or a dynamic array of records. Arrays cannot be composed of Ingres data types.

The following menu operations are available for this pop-up frame:

OK

Creates the record attribute

Cancel

Cancels creation of the record attribute

ListChoices

Displays a list of legal values for the field from which you can choose *or* field data specifications

Help

Accesses the Help utility

Define a Record Attribute

From the Edit a Record Type Definition catalog frame, shown in the Edit a Record Type Definition catalog frame figure, specify the attributes of the record type you have created.

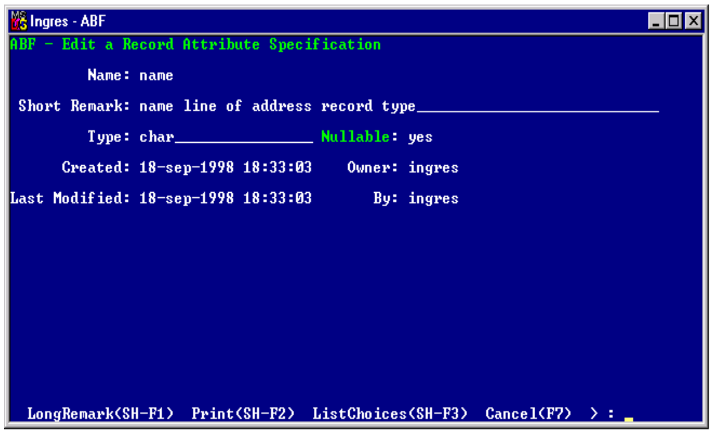
To define a record attribute

1. From the Edit a Record Type Definition frame, select Create. This displays the Create a Record Attribute frame, shown in the previous figure.
2. In the required Name field, enter a name for the attribute. This cannot be the name of another attribute for this record.
3. In the required Data Type field, enter the name of any record type defined for this application or an Ingres type.

   For help on this field, select the ListChoices menu item. This displays a menu-type list of values from which you can select.
4. Depending on which data type you enter, the next field is either the Array or the Nullable field. For more information, see The Create a Record Attribute Frame section.
5. You can optionally enter and modify the Short Remark field here.
6. Select OK to create the attribute. You see the Edit a Record Attribute Specification frame.

The Edit a Record Attribute Specification Frame

Use the Edit a Record Attribute Specification frame, shown below, to define or edit the attributes that make up a record you have created. Call this frame by selecting the Edit menu operation from the Edit a Record Definition frame.



```
ABF - Edit a Record Attribute Specification

         Name: name

 Short Remark: name line of address record type____________________________

         Type: char________________ Nullable: yes

      Created: 18-sep-1998 18:33:03    Owner: ingres

Last Modified: 18-sep-1998 18:33:03       By: ingres

  LongRemark(SH-F1)  Print(SH-F2)  ListChoices(SH-F3)  Cancel(F7)  > :
```

The fields in this frame contain the information you entered in the Create a Record Attribute Frame.

Name

(Read-only) Specifies the name of the record attribute

Short Remark

(Optional) Provides a brief description

Type

(Required) Specifies the name of any record attribute defined for this application or any Ingres type. The data type entered here determines whether the Nullable or the Array field is active for the attribute.

- If Type is an Ingres data type, the Nullable field is active.
- If Type is a record type created for the application, the Array field is active.

Nullable

Indicates whether the record attribute is nullable. Only attributes of an Ingres type can also be nullable. Record types cannot be nullable. The default depends on the application query language: "yes" for SQL and "no" for QUEL.

Array

Indicates whether this attribute is a single record or a dynamic array of records

The remaining fields contain information about when the attribute was created and last modified, and the users who created or modified it.

The following table describes this frame's menu operations:

LongRemark

Reads or edits more information about the record attribute on the Long Remark pop-up frame

Print

Prints the text file containing the 4GL specifications

ListChoices

Displays a list of legal values for the field from which you can choose *or* field description

Cancel

Returns to the previous frame without saving changes

Help

Displays help for this frame

End

Returns to the previous frame. Saves any changes made to the record definition and then returns you to the Edit a Record Definition frame. All field values must be valid before this is allowed.

Edit a Record Attribute

After you create a record attribute, you see the Edit a Record Attribute Specification frame, as shown in the previous figure. The Name field shows the attribute name you entered in the Create a Record Attribute frame.

To edit a record attribute

1. (Optional) Enter a brief description in the Short Remark field. Select LongRemark to enter more details about the attribute on the LongRemark pop-up.
2. In the required Data Type field, enter the name of any record type defined for this application or any
   Ingres type.

   For help on this field, select the ListChoices menu item. This displays a menu-type list of values from which you can select.
3. Depending on which data type you enter, the next field is either the Array or the Nullable field. For more information, see The Create a Record Attribute Frame (see page 416).
4. Ensure that all field values are valid. Select End to save the attribute description and exit to the Edit a Record Definition frame.

Compatibility with Future Record Data Types

Future releases of the Ingres DBMS system can include Ingres types whose names conflict with record types previously defined by the user. Ingres allows for this situation by continuing to support previously defined record types.

Whenever a data type can be either a record type or an Ingres data type, Ingres searches the record types first, followed by the data types. However, users cannot define new record types conflicting with new or existing Ingres data types.

Selecting a Text Editor

You can specify any editor available on your computer system with the ING_EDIT option. This feature is valuable if you must edit frames and procedures called by your application and use a different editor. To use a text editor other than the default, you must redefine the ING_EDIT option before you start ABF.

Windows: For example, to define the text editor as "edit":

```
set ING_EDIT=edit
```

The default text editor is notepad unless your system administrator has set a different system-wide default when installing Ingres.

UNIX: For example, to define the text editor as "vi":

```
setenv ING_EDIT /usr/ucb/vi
```

The default text editor is "ed," unless your system administrator has set a different system-wide default when installing Ingres.

VMS: Defining the logical ING_EDIT as "+EDT," "+LSE," or "+TPU" sets your default editor to the callable version of the editor that you select. Using the callable editor avoids creating a new subprocess to run the editor; therefore, the editor starts up more quickly.

The callable editor names do not take any qualifiers. Specify any non-default editor behavior in editor startup files.

For example, to use the Language Sensitive Editor in Fortran:

```
$ DEFINE ING_EDIT "LSE/LANGUAGE=FORTRAN"
```

To use the callable version of the Language Sensitive Editor:

```
$ DEFINE ING_EDIT "+LSE"
$ DEFINE LSE$COMMAND "SYS$LOGIN:LSE_STARTUP.LSE"
```

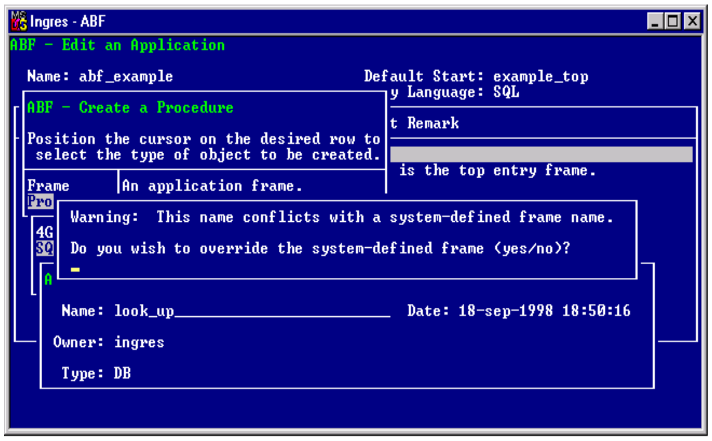
In the previous example, SYS$LOGIN:LSE_STARTUP.LSE includes the command "SET LANGUAGE FORTRAN."

The default text editor is "EDT."

Built-in Frames and Procedures

ABF provides built-in frames and procedures that you can call with 4GL code.

You can override these frames and procedures by creating frames or procedures with the same names. ABF displays a pop-up prompting you to verify whether you want to override the built-ins. This is similar to the warning that ABF displays when the name of a procedure conflicts with a system function. An example of this message and prompt is shown in the following figure:



```
ABF - Edit an Application

 Name: abf_example                        Default Start: example_top
                                         y Language: SQL
 ABF - Create a Procedure
                                         t Remark
 Position the cursor on the desired row to
  select the type of object to be created.
                                           is the top entry frame.
 Frame       An application frame.
 Pro
      Warning:  This name conflicts with a system-defined frame name.
 4G
 SQ   Do you wish to override the system-defined frame (yes/no)?

   A

      Name: look_up________________________   Date: 18-sep-1998 18:50:16

     Owner: ingres

      Type: DB
```

The built-in frames and procedures available for your ABF applications are summarized below. For details about how each works and how to use it, look under the procedure or function name in the 4GL section of this guide.

Lookup frame

The look_up() frame lets you implement a user-specified lookup of values from the database with a single call. Results appear in the window in a menu-type pop-up list.

Sequence values

The sequence_value() procedure returns a new sequence value (or range) for a 4GL database column.

Find record column value

The find_record() procedure prompts the user for a value to find in the column of the current table field and scrolls to that row if the value exists.

Provide help on a field

The help_field() procedure provides help on a field consisting of either a description of the validation of a field or a pop-up list of values.

Ring monitor bell

The beep() procedure causes the monitor to beep or ring, if the monitor has that capability.

Clear the records from an array

The ArrayClear() procedure removes all array records, including any marked Deleted.

Count records in an array

The ArrayAllRows() procedure returns the total number of records in an array, including those marked deleted.

Count non-deleted records in an array

The ArrayLastRow() procedure returns the number of non-deleted records.

Return the number of the first deleted row in an array

The ArrayFirstRow() procedure returns the number of the first deleted row in an array.

Insert a record into an array

The ArrayInsertRow() procedure inserts a record before the record number you specify and renumbers the records that follow it.

Delete an array record

The ArraySetRowDeleted() procedure marks the specified record deleted.

Remove a record from an array

The ArrayRemoveRow() procedure permanently removes the specified record from an array.

Retrieve application specific parameters

The CommandLineParameters() procedure retrieves parameters that you enter after the -a flag on the command line when you invoke an imaged application.

Working with Existing Frames or Procedures

In addition to creating frames and procedures, you can modify them, destroy them, and run them in complete or incomplete form through Applications-By-Forms.

Modify Frames or Procedures

After you have created a series of frames and procedures for an application, you can modify them easily.

To modify a frame or procedure

1. At the Edit an Application frame, highlight the frame or procedure you want to modify and choose Edit.
2. When you see the appropriate Edit a Definition frame, choose the edit option you need. FormEdit allows you to edit forms. Edit runs the system or report editor for report specifications, host language source code, and 4GL-source code.
3. Choose Compile after editing forms or source code procedures to incorporate your changes into the application.

After each editing session, ABF updates the modification date on the related Edit a Definition frame. When you make changes to a form, ABF automatically marks the 4GL specification for the frames that use that form for recompilation. They are recompiled when you select Go or Image for the frame or application.

Rename Frames or Procedures

When you rename a frame or procedure, perform the rename and then change any default names or other calls to that specific frame or procedure.

To rename a frame or procedure

1. At the Edit an Application frame, move the highlight to the frame or procedure to be renamed. Choose Rename.
2. ABF prompts you for the new name.
3. Enter the name and press Return.

Destroy Frames or Procedures

To eliminate part of an application, destroy each frame or procedure for that part. You must also change any frame that calls the ones you have destroyed.

To destroy a frame or procedure

1. At the Edit an Application frame, move the highlight to the frame or procedure to be destroyed.
2. Select Destroy from the menu.

   ABF prompts you to verify the Destroy operation.
3. Select y (yes) and press Return to remove the item.

Destroying a frame or procedure does not destroy its source code file or form. You can use the same source code file or form for another frame or procedure in this or some other application.

Important! The Destroy operation also appears on the Application Catalog frame. Be very careful not to destroy the application when you want to destroy only a frame or procedure.

Testing Incomplete Applications

You can test the application at any time during development by choosing Go from the Applications Catalog frame or the Edit an Application frame, or from any frame.

- The Go operation is a quick way to check whether the frames already in the application are working the way you expect before you have added all the frames and procedures.
- The Go operation recompiles the parts of the application that you have modified since the last run, and, if there are no errors, executes the application.

The Go operation compiles all frames and source code files needed for the test into the database catalogs. For example, if frame A calls frame B, and frame B calls frame C, a test of A checks A, B, and C for recompilation, but a test of B only checks frames B and C.

To test a 3GL procedure that receives parameters from an ABF frame, select Go from the calling frame. Do not select Go from the 3GL procedure, because this has unpredictable results, suc as the 3GL procedure not working properly or your system returning an Access Violation.

If the application runs under a role, ABF prompts you for the role's password. For security reasons, the password is not displayed on your terminal screen as you type it in. ABF prompts for the password twice. Once you enter a password for an application, you are asked whether you want to use the same password again.

If there are no errors, Go then runs the application, prompting you for the frame to call first. If you select Go from a completed frame, ABF asks if it must start at this frame.

In contrast to the Go operation, you can use Compile to compile a single frame and correct any errors, but Compile does not run the frame. To run, your current frame and all non-4GL procedures must be error-free.

When you choose the Go operation, ABF cannot check the dates on libraries included by the linker options file. For the purposes of building an interpreter, ABF assumes that no modules in the linker file are updated during the current editing session. To link any modules that were updated during the current session, the user must exit from ABF, or go at least as far as the Edit an Application frame, and re-enter the application using the Edit or Go operation. This forces the updated modules to be linked.

How the Test Application Is Built: Procedures

On most systems, as a default, the Go operation runs an application interpretively. This does not include host-language procedures, which are maintained in compiled form and linked into the test application.

If no host-language procedures are defined for the application, the Go operation runs the application using a special interpreter which it places in a file in the Ingres system binary directory, named:

Windows: %II_SYSTEM%\Ingres\bin\iiinterp

UNIX: $II_SYSTEM/ingres/bin/iiinterp

VMS: II_SYSTEM:[INGRES.BIN] iiinterp.exe

If host-language procedures are defined for the application, the Go operation runs the application using a separate interpreter that it creates and places in the application's object code directory. This interpreter file is called:

Windows: iiinterp

UNIX: iiinterp

VMS: iiinterp.exe

How ABF Handles Dependencies

ABF handles dependencies among frames and other application components by performing several types of checking as you develop an application. Based on this checking, ABF determines whether to automatically recompile a frame the next time the application is built.

In the first type of checking, ABF keeps track of the components on which a frame depends. If any component changes, ABF automatically recompiles the frame at the next application build. This happens in the following four types of dependencies:

- A frame calls another frame and gets a return value from that frame. If the return type of the second frame changes, the first frame is recompiled.
- A frame uses a global variable. If the global variable changes, ABF recompiles the frame.
- A frame has a variable of a record type. If the definition of the record type changes, ABF recompiles the frame.
- A frame uses variables of a type of form or type of table. If the underlying form or table changes, ABF recompiles the frame.

ABF does not compile all called frames or procedures. In the following cases, ABF cannot tell the name of the called frame or procedure at compilation time so the called frame or procedure cannot be compiled:

- When you use a variable to represent a called frame or procedure in a callframe or callproc statement. For example, the frame represented by the variable *nextframe* is not compiled:

  ```
  callframe :nextframe
  ```

- When a called SQL procedure calls a 4GL frame or procedure using an exec 4GL statement. For example, the procedure "procname" is not compiled:

  ```
  exec 4GL procname
  ```

In the second type of checking, ABF allows you to modify global or record attributes of a record type used in an application, but does not allow you to delete or rename the record type. ABF notices if you have modified the record type and automatically recompiles frames that use the record type.

In the third type of checking, ABF does not allow a record type to contain an attribute of its own type. This is not allowed even if there are steps in between. For example, if record type A includes an attribute of type C, C cannot contain an attribute of record type A.

The fourth type of checking concerns the Go operation. The Go operation involves the frame from which you choose Go (the start frame for the test build) and all of its children. Two types of components must be error-free, the start frame and any 3GL procedures you have written. When you choose Go from a frame that is not the application start frame, ABF checks this frame and all of its children to see if they need recompiling. It does not check anything outside of this line of dependencies.

For example, frame A calls frame B, B calls C. If you run B by selecting Go, you cannot call A. You can go only to children of B, not to the parent frame A. When ABF checks that frames must be recompiled, it checks B and all of B's descendants.

For more information on callframe and writing 4GL specifications, see the 4GL section of this guide.

Creating an Image of the Application

An *executable image* is a compiled version of the application that the user can run directly from the operating system, without going through ABF. The image file is a "snapshot" of the application at the time the image is made. Creating an executable image of your application gives you three advantages:

- *Maintaining the application's integrity*. Your application is intended as a program that end users run to do their work. If users run it from within ABF, they have the ability to redefine the application. Problems arise if accidental changes make their way into the application.
- *Optimizing the application's performance.* Better system performance results when the application runs from the operating system level and does not include the overhead of starting up ABF.
- *Running the image against a different database.* If you have a development and a production version of the same database, you can develop and image the application in the development database and run it against the production database. See the section, Running the Application on a Different Database.

When you create an executable image of the application, ABF:

- Ensures that the compiled form file, which has a .c file extension (.mar for VMS), for each compiled form in the application is up to date
- Compiles it into an object file (.obj) in the object code directory
- Links this object code file into the executable
- Does not look for later versions of the form in the system catalog

Building an executable image can be a lengthy process, often taking as much as five times longer than running the application with the Go operation. This is because the frames and procedures defined in the application must be compiled and linked.

On the other hand, the program is more convenient to run from the image than under the Go operation. See Testing Incomplete Applications (see page 426) for more information about the behavior of the Go operation.

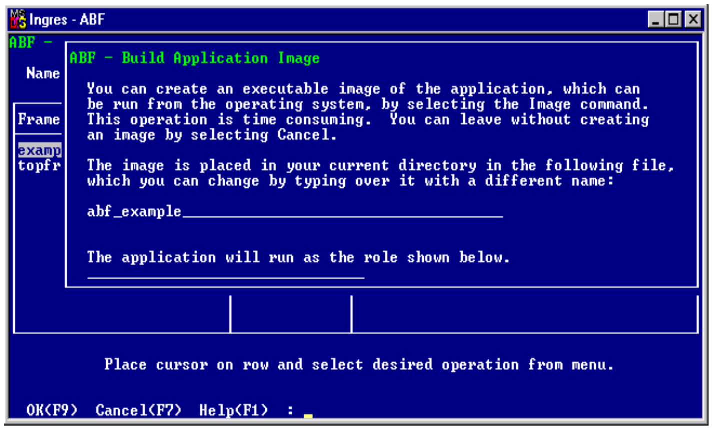
Build an Image from Within ABF

You can create and run an executable image of an application while you are developing it.

To create and run the application image

1. Debug all the frames and procedures in the application, including the reports and forms.
2. Select Image from the Utilities menu of the Edit an Application frame.

   The Build Application Image pop-up frame appears, as shown in the following figure:



```
ABF - Build Application Image

 You can create an executable image of the application, which can
 be run from the operating system, by selecting the Image command.
 This operation is time consuming.  You can leave without creating
 an image by selecting Cancel.

 The image is placed in your current directory in the following file,
 which you can change by typing over it with a different name:

 abf_example_____________________________________

 The application will run as the role shown below.
 ________________________________

          Place cursor on row and select desired operation from menu.

 OK(F9)  Cancel(F7)  Help(F1)  :
```

   This frame includes messages, the default name of the image file, and the application role.

3. You can change the image name or the role name as follows:
   - The image file name is either the default image name specified in the Application Defaults frame or, if none is specified, the name of the application plus a file extension (if required by the operating system). You can change the name here. The new name must be a legal file name and can contain up to 40 characters. This does not change the default image name on the Application Defaults frame.
   - If the application runs under a role, ABF prompts you for the role's password during the Image operation with a pop-up frame. The password is not displayed on your terminal screen as you type it in. ABF prompts for the password twice. For more information on application roles, see Application Defaults Frame (see page 358) and Roles for ABF Applications (see page 537). Roles are a feature of the Knowledge Management Extension.

   After creating an image, you can specify one or more symbols to use to start the application from operating system level. See Defining Symbols for Different Users (see page 438).
4. Select OK on the Build Application Image frame.

   ABF displays:
   - Compilation status and error messages

     If there are errors in the 4GL or 3GL code, ABF stops building the image. You can see the errors by displaying the Error Listing Frame.
   - Application linking messages to indicate any undefined procedures or variables
5. When the executable image is complete, select End to return to the Edit an Application frame.

If an application uses library host language procedures, you must link it to the appropriate host language library.

To link your application to a host language library

1. Create a linker options file containing the host language library.
2. Define the linker options file to ABF using either the ABF Application Defaults frame or ING_ABFOPT1

   The ABF Default setting overrides the setting of ING_ABFOPT1. For details on using ING_ABFOPT1, see ABF Architecture (see page 501).

The Image operation always relinks an application. This means that if modules in the linker file are updated during the current editing session, Image links with the most current modules available.

imageapp Command—Build an Image from the System Level

You can build an application image from the operating system level with the imageapp command, described in this section. This command (called abfimage in earlier releases) builds an executable image, like the Image operation from the ABF menu, but uses a command line syntax instead of a forms-based interface. You can include the imageapp command in a command file.

Errors found during the image build are reported on the standard error device. The .lis file in the object code directory contains detailed descriptions of compilation errors (as described in ABF Architecture (see page 501)).

This command has the following syntax:

```
imageapp [v_node::]dbname applicationname [-uusername][-f]
    [-w][-5.0][+wopen][-oimagename][-GgroupID]
    [-Rrolename]
    [-constants_file='filename']
```

[*v_node*::]*dbname*

Specifies the name of the database.

If you are using a database that resides on a remote host, you must specify the *v_node,* followed by two colons. For example:

```
server1::orderdb
```

applicationname

Specifies the name of the application

If specified, you enter Vision at the Application Flow Diagram for that application. If not specified, you enter Vision at the Applications Catalog frame.

-u*username*

Runs the application as if you were the user represented by *username.*

Files created under this flag are owned by the user actually running the Vision process, not by *username.*

To use this option, you must be a privileged user.

If you are using Enterprise Access Products (formerly Gateways), see your Enterprise Access documentation before using this parameter.

-f

Forces Vision to recompile the 4GL code for the entire application, even if no changes have been made since the last time it was compiled

-w

Causes all warnings to be turned off. See the section Disabling Warnings for more information.

-5.0

Invokes 4GL in a mode that is compatible with CA-Ingres Release 5. See 5.0 Applications (see page 1383) for more information.

+wopen

Generates warnings if Vision detects statements that are not compatible with OpenSQL

-o*imagename*

Specifies the name for the image. If the -o flag is not specified, Vision uses the default image name specified in the Applications Defaults window. If no default image name is specified, the image is given the same name as the application.

-G*groupID*

VMS: Lets you run or edit the application as a member of the group specified.

Capital letter flags require double quotes in VMS, for example:"-G*groupid*"

-R*rolename*

Assigns a role to an application image.

If you specify a role name, you are prompted for the role's password. Roles are a feature of the Knowledge Management Extension.

VMS: Capital letter flags require double quotes, for example: "-R*rolename*"

-constants_file='*filename*'

Specifies a file containing values for the application's constants. If the -constants_file flag is specified, the values in the constants file override the values for the constants stored in the application.

The *filename* can be the full directory path name for the constants file.

Example—imageapp command:

To recompile all frames and create an image called "orders" for the Order_entry application in the Inventory database, enter:

```
imageapp inventory order_entry -oorders -f
```

imagename Command—Run a Completed Application

To run a completed application, use the following command syntax at the operating system prompt:

```
imagename [ -d[v_node::]dbname | -database=[v_node::]dbname ]
    | -nodatabase ] [-uusername] [framename |[-p]procname]
    [-noforms|forms] [SQL option flags] [-Ggroupid]
    [-Rrolename]
    [-constants_file='filename']
    [-a application_specific_parameter
    {application_specific_parameter}]
```

Windows: To run a standard application image, created by running imageapp, use the previous command syntax. If you created an interpreted image, by selecting Image from within ABF or by running imagebld, not all of the command flags are supported. Use the following syntax:

```
imagename [ -d[-database][v_node::]dbname | -nodatabase ]
    [-uusername] [-fframename |[-p]procname]
    [-constants_file='filename']
    [-a application_specific_parameter
    {application_specific_parameter}]
```

imagename

Specifies the name of the application image. By default, the image name is the same as your application name.

-d[*v_node*::]*dbname* |

-database[*v_node*::] *dbname*

Runs the application with the database specified by *dbname*.

The -database flag can be abbreviated -d.

The -database or -d flag lets you run the application with a database other than the one the database resides in. The new database must contain the same tables and reports as those used in the application's queries.

For example, you can develop an application on a test database and later run it on a production database.

If you are using a database that resides on a remote host, you must specify *nodename,* followed by two colons. For example:

```
-dserver1::orderdb
```

-nodatabase

Starts the application without an open database session. The **-**nodatabase flag can be used to run an application that does not require access to a database, or an application that starts a database session with the 4GL connect statement.

See the 4GL section of this guide for information about database connections.

-u*username*

Runs the application as if you were the user represented by *username*.

Files created under this flag are owned by the user actually running the ABF process, not by *username.*

To use this option, you must be a privileged user.

If you are using Enterprise Access Products (formerly Gateways), see your Enterprise Access documentation before using this parameter.

framename

Runs the application with the specified frame as the top frame

[-p]*procname*

Runs the application beginning with the procedure represented by *procname*.

You must include the -p flag before the procedure name if you are invoking an image created under a previous release of Vision and the procedure has the same name as a frame in the application.

-noforms|-forms

The -noforms flag lets you run an imaged application without initializing the Forms Runtime System. If the application attempts an operation that requires the forms system, a runtime error is reported.

The -noforms flag can only be used to run applications that do not require any forms. The application can contain forms, if you run the application in a way that does not call the forms. For example, you can start the application from a procedure instead of a frame by specifying the *procnam*e on the command line.

The message statement does not require the forms system.

The -forms flag to call the forms system is the default and is included only for consistency.

SQL option flags

The SQL option flags are flags that affect the database behavior. Vision passes the flags to the database, which interprets them.

See the sql command description in the *Command Reference Guide* for detailed information on these flags. The following SQL option flags are accepted when you run an image:

-f +U -l -x

-G*groupid*

Lets you run or edit the application as a member of the group specified.

VMS: Capital letter flags require double quotes in VMS, for example: "-G*groupid*"

-R*rolename*

Runs the application image with the role specified.

If you specify a role name, you are prompted for the role's password. Roles are a feature of the Knowledge Management Extension.

VMS: Capital letter flags require double quotes, for example: "-R*rolename*"

-constants_file='*filename*'

Specifies a file containing values for the application's constants. If the -constants_file flag is specified, the values in the constants file override the values for the constants stored in the application.

The *filename* can be the full directory path name for the constants file.

-a *application_ specific_parameters*

Allows the user to pass one or more application-specific parameters to the application.

The -a flag must be the last flag that appears on the command line. There must be a blank space between the -a flag and the first parameter that follows it.

Any characters following the -a flag are passed as a single string of parameters.

You can retrieve the parameter values into the application with the CommandLineParameters() function. See Using Application-specific Parameters (see page 440) for details.

Example—imagename command:

To run the order entry application that you imaged as "orders," enter the following command at the operating system prompt:

Windows:

```
orders
```

The orders file must be in the PATH.

UNIX:

```
orders
```

VMS:

```
run orders
```

or

```
run orders.exe
```

The application runs from the top frame or other default start frame (see Set a Default Start Frame (see page 361)).

To access the application beginning with the frame AddOrders, the user enters:

Windows:

```
orders addorders
```

UNIX:

```
orders addorders
```

VMS:

The imagename must be defined as a DCL foreign command. See the following section for instructions and an example.

You also can let users run the application with a command that you specify as described in Defining Symbols for Different Users (see page 438).

Running the Image from System Level

After you complete an executable image of an application, you can run it from the operating system level. See the details below that apply to your operating system.

Windows: You can create a shortcut to run the application from the desktop. For example, to run the Sales Entry Application in the ABF Development Example chapter, create a shortcut to the following command:

```
full_pathname\sales.exe
```

A user can also run the application from an operating system command prompt, passing parameters as necessary.

For example, to specify the first frame to call, enter either of the following commands at the command prompt:

```
salesapp -u newuser topframe
saleapp -xf topframe
```

For flags, see imagename Command—Run a Completed Application (see page 434). For further information, see your query language reference guide.

Defining Symbols for Different Users

To give the application user the convenience of typing in only one name to run the application, you can create a command with the frame name as part of it.

The following directions designate "sales" as the name that users type in to run the Sales Entry Application and Topframe as the initial frame. To do this, follow the steps for your operating system:

Windows: Create a shortcut, sales, to the following command:

```
full_pathname\sales.exe topframe
```

After this, users must select the sales icon to run the Sales Entry Application.

You can allow users to start the application on different frames. For example, if users must run only the Customer frame and not the Topframe, you can create a shortcut for them that runs Sales by calling the Customer frame with the command cust. Either of the following commands in an icon runs the Sales application by calling the frame Customer:

```
full_pathname\sales customer
saleapp customer
```

UNIX: Define the alias or shell script name:

```
alias sales 'full_pathname/sales.exe topframe'
```

You can define an alias if you use the C shell. If you use the Bourne shell, you must write a shell script.

After this, users must type sales at the operating system level to run the Sales Entry Application.

If you defined saleapp as in the previous section, define "sales" as follows:

```
alias sales 'saleapp topframe'
```

This is possible only if you use the C shell. Bourne shell users must write a shell script.

You can allow users to start the application on different frames. For example, if users must run the Customer frame and not the Topframe, you can define an alias for them that runs Sales by calling the Customer frame with the command customers. Either of the following definitions creates the symbol "customers." When entered, customers runs the Sales application by calling the frame Customer:

```
alias customers 'full_pathname/sales customer'
alias customers 'saleapp customer'
```

VMS: To designate "sales" as the name that users type to run the Sales Entry Application by calling Topframe, define the symbol:

```
sales :== "$dir_spec sales.exe topframe"
```

After this, users must type sales at the operating system level to run this application.

If you defined saleapp as in the previous section, define "sales" as follows:

```
sales == saleapp + " topframe"
```

You can allow users to start the application on different frames. For example, if users must run the Customer frame and not the Topframe, you can define an alias for them that runs Sales by calling the Customer frame with the command customers.

Either of the following definitions creates the symbol "customers." When entered, customers runs Sales by calling the frame Customer:

```
customers :== "$dir_spec sales.exe customer"
customers == saleapp + " customer"
```

Using Application-specific Parameters

When users run an imaged application from the command line, you can let them specify values for parameters that are passed back to the application itself. For example, you can ask the user to indicate a department name when starting the application. You then can use this value to restrict the records that the user sees on various frames of the application.

The -a flag indicates the start of a string of application-specific parameters. You retrieve the values into the application with the CommandLineParameters() function. This function returns a varchar value of the appropriate size (up to *n* bytes, where *n* represents the lesser of the maximum configured row size and 32,000 to hold the parameters specified on the command line (up to 126 characters for PCs). After *n* bytes, any additional characters are truncated.

VMS: The parameter list can be up to 512 bytes, including up to 256 parameters.

Note: The -a flag must be the last flag on the command line.

In an application, you can define as many variables as you must hold the values that the user specifies. You then can use these variables any number of times, from any frames or procedures within the application.

If you want to use multiple parameters, be aware that the CommandLineParameters() function always returns a single value—all the parameters concatenated into a single string, with the parameters separated by a single space. You must write your own code for parsing this value into the separate variables for your application to use. The characters that the user enters after the -a flag, including multiple spaces, can be parsed differently by different operating systems.

See the 4GL section of this guide for more information on how this function retrieves parameter values, including an example of 4GL string-parsing code.

Destroying an Application

You can destroy an unwanted application from the Applications Catalog frame. Move the highlight to the desired application and choose Destroy. ABF prompts for confirmation. However, ABF does not destroy the forms and source code files. At the operating system level, delete the source code files used by the appropriate command for your operating system.

When you destroy an application, ABF removes the definition of the application from the Extended System catalogs. For information about the system catalogs, consult your query language reference guide.

To facilitate the development of applications in a multi-user environment, the normal Ingres namespace does not apply to ABF applications and frames. This means that all users with access to a database have equal access to applications and frames in that database.

Important! Any user with access to a database can edit or drop any other user's application and frames in the database, and any user can create a frame in any application

Copying an Application into a Different Database

To copy an application from one database to another, use the copyapp command. The copyapp command copies information about an application as stored in the Ingres system catalogs. If the -s flag is specified, copyapp also copies the source code files for the application. For Vision applications, copyapp -s only copies source code files for Custom frames, not Vision-generated frames.

The copy process involves two steps:

- copyapp out

  This command transfers application data from the old database to an intermediate text file. Do not edit the intermediate file, because the application data is stored in a fixed order.

- copyapp in

  This command transfers application data from the text file to the new database.

The copyapp command notifies you if components in the new and old databases have the same names. By default, copyapp does not copy the component if there is a name conflict. If you use the -r flag, copyapp replaces the component in the new database with the same-named component from the old database.

Similarly, you cannot use the copyapp command to merge applications. You can replace an existing application by using the -r flag. By default, the copyapp in operation does not copy the application if an application with the same name exists in the database.

The copyapp command does not transfer any other database components, such as rules, reports, or tables. In order for your application to work correctly, you must be sure that the database contains the correct components.

You can copy individual components of an application—such as forms and reports—from one database to another using copyform, copyrep, and similar utilities. To copy database tables, you can use the copydb and unloaddb utilities. See the *System Administrator's Guide* for more information about any of these utilities.

The following sections describe the syntax of the two steps of the copyapp operation.

Copying Applications to a Different Location

If you are copying an application to a database in a different location than the current database, you must:

- Use -s or -a to copy an application to another directory.

  If you do not use -s or -a with the copyapp in command, change the source code directory path in the Application Defaults window, before or after running copyapp.

- Change any hard-coded path names that appear in 4GL frames by editing the frame's source code.

copyapp out Command—Copy an Application Out of a Database

To copy an application *out*, issue the following command at the operating system prompt.

This command has the following syntax:

```
copyapp out [-ddirname] [-tfilename] [-uusername]
     [-lfilename] olddbname applname
```

-d*dirname*

Indicates that the intermediate file must be placed in the specified directory. The default is the current directory.

-t*filename*

Indicates a name for the intermediate file. If the flag is not set, the intermediate file is named *iicopyapp.tmp*.

-u*username*

Requests that ABF pretend you are the user with login name *username*. You must be a privileged user to use this flag.

If you are using Enterprise Access Products (formerly Gateways), see your Enterprise Access documentation before using this parameter.

-l*filename*

Creates a file containing a list of the names of the source files. For Vision applications, the list includes only Custom frames.

olddbname

Specifies the name of the database with the application

applname

Specifies the name of the application

copyapp in Command—Copy an Application into a Database

To copy an application into a database, issue the following command at the system prompt.

This command has the following syntax:

```
copyapp in [-c] [-ddirname] [-nnewapplname] [-p] [-q] [-r]
     [-s[dirname]|-a[dirname]] [-uusername]
     newdbname intfilename [-lfilename]
```

-c

Deletes (cleans up) the intermediate file

-d*dirname*

Indicates that the intermediate file is in the named directory. Provide the full directory name. Default: the current directory.

-n*newapplname*

Gives the application the specified new name in the new database. Default: keep the same name.

-p

Suppresses messages about name conflicts. Default: issue message upon finding a name conflict.

-q

Causes copyapp to quit if there is any name conflict.

When this flag is set, no changes are made to the database if a name conflict is found.

If you specify the -q flag, copyapp is performed as a single transaction. The copyapp transaction locks system catalogs; for this reason, you must not specify -q when users are connected to the database. In addition, the transaction is logged in the log file; you must be sure that the log file is large enough to accommodate the copyapp transaction.

If neither -q or -r is specified and a duplicate name is encountered, the copy is not done and terminates with an error message. In this case, the application can exist in the new database in a partially copied state.

-r

Overwrites components with the same name

If neither -q or -r is specified and a duplicate name is encountered, the copy is not done and terminates with an error message. In this case, the application can exist in the new database in a partially copied state.

-u*username*

Requests that ABF pretend you are the user with login name *username.* You must be a privileged user to use this flag.

Transfers ownership of the application and application components to username.

If you are using Enterprise Access Products (formerly Gateways), see your Enterprise Access documentation before using this parameter.

-s[*dirname*]

Specifies a new directory for source files. If a directory name is not specified, the current directory is used.

For ABF applications, copies 4GL source files.

For Vision applications, copies only 4GL source files for Custom frames. For *non-custom* Vision frames, copyapp transfers no files. The frames are marked as *new*. The source files for these frames are regenerated at the next Image or Go operation.

Do not use this flag with the -a flag.

-a[*dirname*]

Specifies a new source directory name for the application, but does not transfer source files. If a directory name is not specified, the current directory is used.

This flag marks Vision (non-custom) frames as *new.* The source file is regenerated at the first Image or Go operation.

Do not use this flag with the -s flag.

newdbname

Specifies the name of the database to be copied into

intfilename

Specifies the name of the intermediate file

-l*filename*

Creates a file containing a list of the names of the source files. For Vision applications, the list includes only Custom frames.

If -s is specified the source files in this list were copied. If -a is specified, the source files in this lists were not copied.

As an example of the copyapp command, suppose you want to move the Sales application from the Accounts database into the Newdb database. The following statement performs the first part of the task, copying the application into a text file called *sales.tmp*:

```
copyapp out -tsales.tmp accounts sales
```

Once copyapp copies the application into the *sales.tmp* file, you can copy *sales.tmp* into the Newdb database by executing the following command:

```
copyapp in newdb sales.tmp
```

Copying Application Components

You can copy components of an application to another application, called the target application, using the iiexport and iiimport commands. Unlike the copyapp command, which copies an entire application, iiimport and iiexport copy individual components.

The components you can copy are frames, procedures, records, global variables, and global constants. You can copy all the components of a particular type using one of the -all flags:
-allframes, -allprocs, -allrecords, -allglobals, -allconstants.

Copying components requires two steps:

1. Use iiexport to extract the components to an intermediate file.
2. Use iiimport to load the components from the intermediate file into the target application.

By default, the intermediate file is called iiexport.tmp. You can change the name of the intermediate file with the -intfile flag. If the applications are not on the same machine or virtual file system, you must copy the intermediate file to the target machine before running iiimport.

Using the iiimport and iiexport commands allows you to merge components into an existing application. For example, if you are working on a development team, the team can have one master application. Each team member can develop frames and other components independently, and use iiimport and iiexport to copy the components into the master application.

Use the intermediate file created by iiexport as a snapshot of a frame or other component at a particular stage of development. For example, you can check the file into a source control system or store it as a backup file.

See iiexport Command—Copy Application Components to a File (see page 447) and iiimport Command—Import Application Components (see page 449) for detailed syntax for these commands.

How Dependencies Are Handled

This section describes how iiimport and iiexport handle the relationships between frames and dependent components when you copy a frame.

A frame in an ABF application is interconnected with other frames in the application. For example, a frame can call one or more child frames.

The parent/child structure of the target application can be different from the structure of the exporting application. For the frame to be consistent and usable after the frame is imported, the dependent components must reflect the structure of the target application.

The iiimport command does not change the following dependent components:

- Child frames (unless specified explicitly)

  If the frame is a parent frame in an ABF application, iiimport does not copy the child frames unless you explicitly specified the child frames to iiexport.

- Global components (unless specified explicitly)

  A frame can also be dependent on global components, that is, global constants, record types, global procedures, or global variables. For the frame to run correctly, those components must exist in the target application. When you copy a frame with iiexport and iiimport, global components are not automatically copied, even if the frame references them. To copy global components, you can specify them explicitly to iiexport.

iiexport Command—Copy Application Components to a File

The iiexport command copies one or more application components to a file. A component can be a frame, procedure, record type, global variable, or a global constant. You can copy all the components of a particular type from an application using one of the -all flags: -allframes, -allprocs, -allrecords, -allglobals, -allconstants.

When you copy a frame, iiexport copies the frame definition, including local components and Escape code. However, iiexport does not copy global components or child frames, unless they are explicitly specified. See the Handling Dependencies section for more information.

This command has the following syntax:

```
iiexport dbname applname [-intfile=filename] [-listing=filename]
  [-user=username] component=name{,name} {component=name{,name}}
```

where *component* is -frame, -proc, -record, -global, -constant, -allframes, -allprocs, -allrecords, -allglobals, or -allconstants, and *name*{,*name*} is a comma-separated list with no spaces.

dbname

Specifies the database from which the component is being copied

applname

Specifies the application from which the component is being copied

-intfile=*filename*

Specifies the intermediate file created by iiexport. The *filename* can be the full pathname of the file. If you specify the
-intfile flag, you must indicate a file name. If you do not specify the -intfile flag, iiexport names the intermediate file iiexport.tmp.

-listing=*filename*

Specifies a file or lists the names of the source files for each frame copied. If you specify the -listing flag, you must indicate a file name. If -listing is not specified, ABF does not create a listing file.

-user=*username*

Runs the command as the user specified by username.

To use this option, you must be a privileged user.

If you are using Enterprise Access products (formerly Gateways), see your Enterprise Access documentation before using this parameter.

-frame=*name*{,*name*}

Indicates the name of one or more frames to be copied

-proc=*name*{,*name*}

Indicates the name of one or more procedures to be copied

-record=*name*{,*name*}

Indicates the name of one or more record types to be copied

-global=*name*{,*name*}

Indicates the name of one or more global variables to be copied

-constant=*name*{,*name*}

Indicates the name of one or more global constants to be copied

-allframes

Indicates that all frames must be copied from the application

-allprocs

Indicates that all procedures must be copied from the application

-allrecords

Indicates that all record types must be copied from the application

-allglobals

Indicates that all global variables must be copied from the application

-allconstants

Indicates that all global constants must be copied from the application

iiimport Command—Import Application Components

The iiimport command copies application components from an intermediate file into an existing application. The intermediate file, created by iiexport, contains the definitions for the components you specified when you ran iiexport. See the The iiexport Command section for the types of components you can copy.

When you import a component, the iiimport command assumes that the component exists in the application, and revises the existing version of the component. When you import a frame, the existing frame definition, including local components, is replaced with the new definition. The iiimport command does not change menu items for a frame, except for user-defined menu items included with Escape code. See How Dependencies Are Handled (see page 446) for information on how iiimport handles menu items and other dependent components.

You can check whether a component exists in the target application by using the -check option. Running iiimport with the -check flag does not actually copy or replace any components, but reports on the status of the components in the target application.

You can import a component that does not exist in the target application, but the target application itself must exist. To copy an entire application into a database, use the copyapp utility.

All components copied in are assigned to the owner of the target application.

This command has the following syntax:

```
iiimport dbname appname [-intfile=filename] [-listing=filename]
     [-user=username] [-check | -copysrc] [-dropfile]
```

dbname

Specifies the database to which the component is being copied

appname

Specifies the application to which the component is being copied

-intfile=*filename*

Specifies the intermediate file to be imported. The filename must specify a file created by iiexport. The filename can be the full pathname of the file. If you specify the -intfile flag, you must indicate a file name. If you do not specify the -intfile flag, iiimport assumes the name of the intermediate file is iiexport.tmp.

-listing=*filename*

Specifies a file that contains a list of source code files for the copied frames or procedures

If you specify the -copysrc flag, the source files listed in this file are the files iiexport/iiimport copies.

If you do not specify the -copysrc flag, you can refer to the listing file to find out which files must be copied manually.

If you specify the -listing flag, you must indicate a file name. If -listing is not specified, ABF does not create a listing file.

-user=*username*

Runs the command as the user specified by the username.

To use this option, you must be a privileged user.

If you are using Enterprise Access products (formerly Gateways), see your Enterprise Access documentation before using this parameter.

-check

Reports the status of the components before actually replacing or importing any components. Use the -check option to report whether the components in the iiexport file exist in the target application. The status is 0 if all components were found. If not all components were found, the status is a non-zero integer.

-copysrc

Specifies that application's source code files are copied to the target application's source code directory. These are the files named in the -listing file, if one was specified. You cannot use the -copysrc flag if the applications are on different machines.

-dropfile

Deletes the intermediate file when the import operation is completed

Master Application Example

If multiple developers are working on an application, they can develop portions of the application independently. Then they can use iiimport and iiexport to copy frames and other components into a master application as they are developed.

In this example, the master application is the "orders" application. The development team designs the overall structure of the application.

Now the developers are assigned to different pieces of the application. Developer A is assigned to modify "neworders," an Append frame called by the "orders" Menu frame.

Developer A makes the following changes to the application:

- Modifies the "neworders" frame
- Adds the called procedure "calctax"
- Adds the child frame "addbooks"
- Passes some parameters from "neworders" to "changeinfo"
- Passes parameters from "neworders" to "addbooks"

Examples:

The following examples use the scenario described in the section, Master Application Example. Developer A wants to merge the revised frames and other components into the master "orders" application.

- Replace the master "neworders" frame with the "neworders" frame in abc_orders:

  ```
  iiexport tutor abc_orders -frame=neworders
  iiimport tutor orders
  ```

  After the import, the child frames "calctax" and "addbooks" do not appear in the "orders" application, or in the menu for the "neworders" frame in the target application. The "changeinfo" frame is not changed.

- Copy the procedure "calctax" into the master application:

  ```
  iiexport tutor abc_orders -proc=calctax
  iiimport tutor orders
  ```

  If the procedure uses a global variable *taxamt*, the variable must exist in the target application. If it does not, copy the variable by specifying it to **iiexport**:

  ```
  iiexport tutor abc_orders -global=taxamt
  iiimport tutor orders
  ```

  You can also copy the procedure and the variable at the same time:

  ```
  iiexport tutor abc_orders -proc=calctax -    global=taxamt
  iiimport tutor orders
  ```

- Copy the frame "neworders" and its child frame, "changeinfo":

  ```
  iiexport tutor abc_orders -frame=neworders,changeinfo
  iiimport tutor orders
  ```

  In this case, because the child frame "changeinfo" is specified to iiexport, the "changeinfo" frame in the target application is changed.

- Copy the frame "neworders" and its child frame "addbooks":

  ```
  iiexport tutor abc_orders -frame=neworders,addbooks
  iiimport tutor orders
  ```

  In this case, the child frame "addbooks" is specified to iiexport, so the frame is copied.

Running the Application on a Different Database

You can run your application on a different database from the one in which you created it by defining a special command. This feature is useful when you must develop an application on a *test database* and later run the application on a *production database*. You can do this in either of the ways described below.

In the first method, the Sales Entry Application remains in the database in which you developed it. You developed the Sales Entry Application on the test database (Testdb) and created an executable image (*sales.exe*). Testdb is the only database containing a definition of the Sales Entry Application.

To use the Sales Entry Application on the Proddb database, make sure that all the components in the application—forms, reports and tables—also exist in the production database (Proddb). If the forms of the Sales Entry Application have been linked into the application image, it is not necessary to copy the forms. Reports are never linked in, and so must always be copied. Perform the following instruction that applies to your operating system:

Windows: Make a shortcut called salesapp to the following command:

```
salesapp 'full_pathname\sales -dnodename::proddb'
```

UNIX: Execute the following command at your operating system level:

```
alias salesapp 'full_pathname/sales -dproddb'
```

VMS: Execute the following command at your operating system level:

```
salesapp :== "$dir_spec sales.exe -dproddb"
```

After this, users can execute salesapp to run the Sales application in the Proddb database, even though the definition of the Sales application resides in the Testdb database.

As an alternate approach, copy the entire application from the Testdb database to the Proddb database using the copyapp utility. Then build a new executable image in the Proddb database.

Chapter 14: ABF Development Example

This section contains the following topics:

This chapter takes you through the steps in developing a sample application with ABF. To keep the presentation brief and clear, the Sales Entry Application contains a simple example of each major screen type. The application includes the built-in procedures beep() and help_field().

Follow the development process presented here to see how you can use ABF, with a minimal amount of 4GL coding, to produce a functioning forms-based application.

- The first part of this chapter describes the application's frames and capabilities.
- The second part of the chapter gives detailed directions for creating the application frames.

As you move through this chapter, see the detailed descriptions of the ABF frames and menu operations in Building Applications (see page 351). This chapter avoids repeating the material in that chapter, where possible.

For more information on using 4GL in ABF applications, see the 4GL part of this guide.

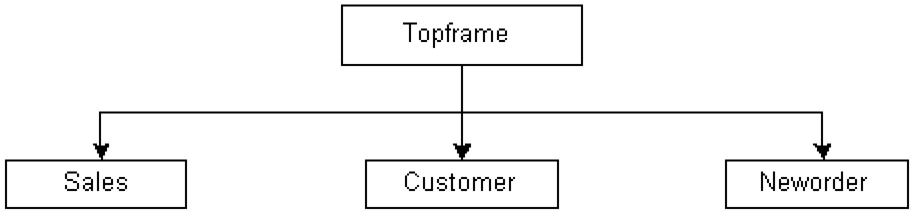
The Sales Entry Application

The Sales Entry Application is designed to help data entry personnel keep track of sales records. The frames include:

- A top frame which serves as a main menu
- A new order frame for entering customer orders
- A frame for updating customer records
- A pop-up report frame for printing a report on sales

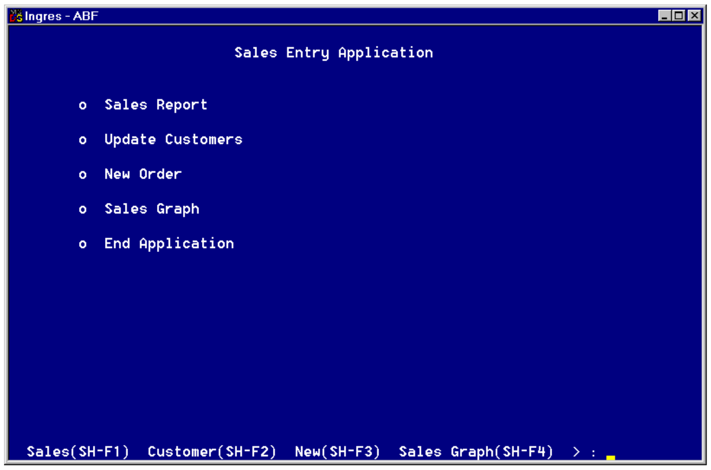
Layout Summary

This chart shows the way frames call other frames in the Sales Entry Application. The boxes in the chart correspond to frames. Arrows indicate that one frame has an operation that can call the second frame. For example, the arrow from Topframe to the box marked Sales indicates that Topframe's Sales operation can call the SaleRep frame.



The Sales Entry Topframe

The first frame, Topframe, shown in the following figure, presents a menu from which users can choose the frames and operations available in the application.



```
                         Sales Entry Application

        o  Sales Report

        o  Update Customers

        o  New Order

        o  Sales Graph

        o  End Application



  Sales(SH-F1)  Customer(SH-F2)  New(SH-F3)  Sales Graph(SH-F4)  > :
```

The Topframe menu operations correspond to the normal daily tasks of the data entry personnel. To select, the user types in the first letter or two of the menu item name or selects the function key automatically assigned by ABF. The following table shows these operations:

Sales

Displays the pop-up Salerep frame so that the user can generate a report on product sales

Customer

Calls the Customer frame so that the user can edit customer records by calling Query-By-Forms (QBF) on the table that contains customer data. The user updates the record using QBF.

New

Calls the NewOrder frame so that the user can enter a new order

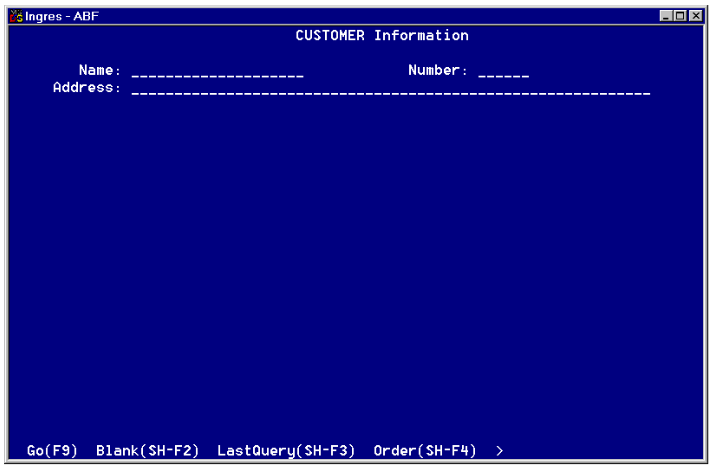
End

Ends the application

The menu item names are set up as global application constants using the Globals menu, to allow for easy substitution or translation to other languages. For more on constants and their use, see Building Applications (see page 351).

The Update Customers Frame

The Customer operation of the Topframe calls a Query-By-Forms (QBF) frame that uses the database table containing customer information. The following figure shows the frame you see when the Update operaton is selected. (For more information on updating, see the chapter, "The QBF Update Operation," in *Character-based Querying and Reporting Tools User Guide*.)



```
                              CUSTOMER Information

       Name: ____________________            Number: ______
    Address: ____________________________________________________________
```

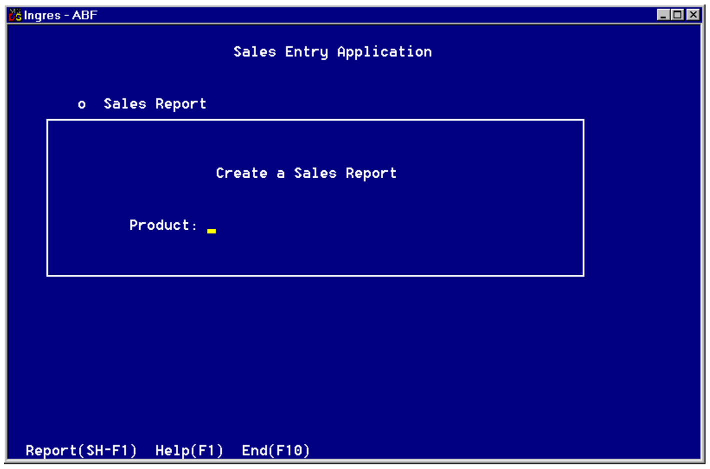
```
Go(F9)  Blank(SH-F2)  LastQuery(SH-F3)  Order(SH-F4)  >
```

The menu is QBF's update menu. The menu changes, displaying different submenus, as the end user specifies different QBF functions. When the user exits QBF, the application returns to the Topframe.

The user must enter customer records in the Customer table through this frame before entering orders in the NewOrder frame. The NewOrder option assumes that any customer placing an order already exists in the Customer table.

The Sales Report Frame

Topframe's Sales Report operation calls the Salerep pop-up frame. The following figure shows Topframe with Salerep displayed. The report presents data on the sales of a particular product. The parameter or variable that determines the content of the report is the name of a product. The user enters this in the Product field. The three Salerep operations displayed on the menu line allow a user to get help on the frame type, run the report, and return to the previous frame.

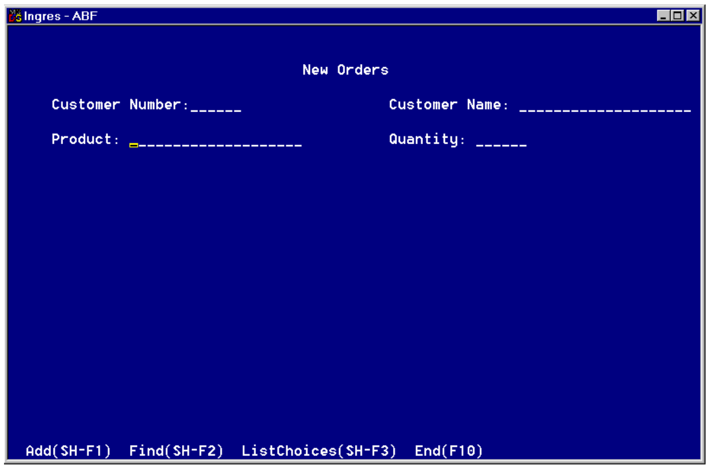


```
Ingres - ABF

                    Sales Entry Application

     o  Sales Report

                 Create a Sales Report

         Product: _

 Report(SH-F1)  Help(F1)  End(F10)
```

The Salerep pop-up is positioned near its menu listing for easy visual association. You can determine the position of the pop-up while creating the form.

The NewOrder Frame

The NewOrder operation calls the NewOrder frame, shown in the following figure. Here the user can add a new order to the database.


Ingres - ABF
New Orders
Customer Number:______
Customer Name: ____________________
Product: ____________________
Quantity: ______
Add(SH-F1) Find(SH-F2) ListChoices(SH-F3) End(F10)

The frame contains the following operations:

Add

Enters a new order. Add appends data to the database and clears the form to allow new values to be entered.

Find

Retrieves information about a customer from the database

ListChoices

Displays a menu of legal values for the current field *or* a field description.

For the Product field, ListChoices displays the list of products that constitute a validation check for that field, specified when building the form. See the Defining the NewOrder Form section for details.

End

Returns to the Topframe

The user selects the Add operation from the NewOrder menu to execute queries that append data to the database. The information is entered in two tables: "Orders" and "Marketing."

- The "Orders" table contains a row for each order.
- The "Marketing" table contains summary information (for example, the number of orders for each product) for tracking product sales.

Defining the Sales Entry Application

This section takes you through starting up ABF, selecting a source code directory, and creating the Sales Entry Application. In the sections that follow, you go on to create tables and frames and test the application. Follow the steps to create your own Sales Entry Application and become more comfortable with ABF.

For more detailed information, see the sections on each procedure in Building Applications (see page 351).

Start ABF

Start up ABF as described in Building Applications (see page 351). This example assumes that the application is defined on a local database called accounts, described below.

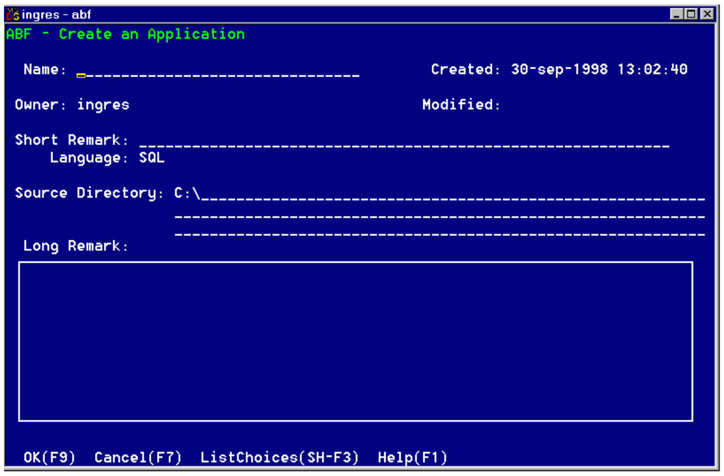
To start ABF

1. Use this command to begin defining a new application:

   ```
   abf accounts
   ```

   ABF starts up on the accounts database, and the Application Catalog frame appears.

2. Select Create. The Create an Application frame appears as in the following figure:



```
ABF - Create an Application

 Name: _______________________________           Created: 30-sep-1998 13:02:40

Owner: ingres                                   Modified:

Short Remark: ____________________________________________________________
    Language: SQL

Source Directory: C:\___________________________________________________________
                  _____________________________________________________________
                  _____________________________________________________________
 Long Remark:


 OK(F9)  Cancel(F7)  ListChoices(SH-F3)  Help(F1)
```

Default values occupy the following fields: the Owner (of the application), Language, Source Directory, and Created (date) fields. Because the Sales Entry Application is new, the other fields are empty.

Create the Application

To create the Sales Entry Application

1. In the Name field of the Create an Application frame, type in sales and press Return.

   The Owner field contains a default user name. Therefore, the cursor moves to the Short Remark field.

2. To enter an optional Short Remark, tab to the Short Remark field and type in Sales Entry Application. Press Return.

3. The Language field contains the default language for the application, as determined by your installation setup. Change this here if necessary or tab past the field without changing it.

4. The Source Code Directory field specifies the directory where ABF stores the application code. The default directory is the directory from which you invoked ABF. You can change the directory here. Make sure that the named directory already exists, and that you have access to it.

   Windows: For example, to store the code in c:\disk\sales, enter **c:\disk\sales** in the Source Directory field.

   UNIX: For example, to store the code in /disk/sales, enter **/disk/sales** in the Source Directory field.

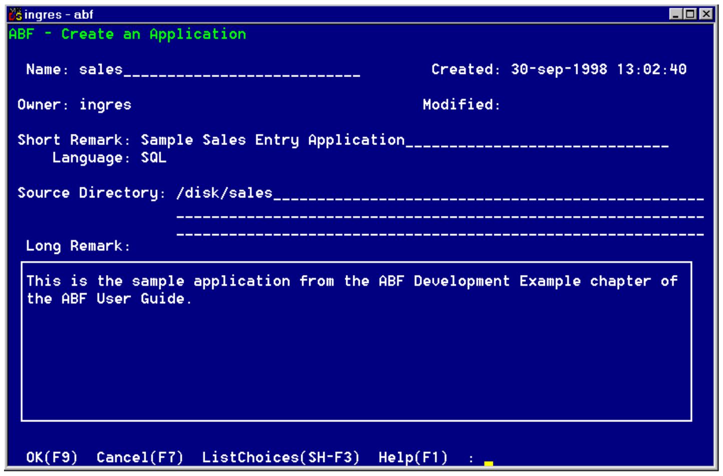
   VMS: For example, to store the code in disk:[sales], enter **disk:[sales]** in the Source Directory field.

   If at some later date you want to change the Source Directory, you can do this by selecting the Defaults menu operation from the Edit an Application frame or the MoreInfo about an Application frame. You must also move all of the application's source code files into the new directory.

5. In the optional Long Remark field, you can enter a longer description of the application. As an example, tab to this field and type:

   ```
   This is the sample application from the "ABF Development Example" chapter of
   the ABF User Guide.
   ```

   The window now appears as shown in the following figure:



```
ABF - Create an Application

  Name: sales___________________________        Created: 30-sep-1998 13:02:40

 Owner: ingres                                 Modified:

 Short Remark: Sample Sales Entry Application______________________________
     Language: SQL

 Source Directory: /disk/sales_____________________________________________
                   ____________________________________________________________
                   ____________________________________________________________
  Long Remark:

  This is the sample application from the ABF Development Example chapter of
  the ABF User Guide.

  OK(F9)  Cancel(F7)  ListChoices(SH-F3)  Help(F1)  :
```

6. Choose the OK operation. ABF appends the necessary entries to the ABF system catalogs to record the creation of the Sales Entry Application.

After creating the application, ABF takes you automatically to the Edit an Application frame.

Guidelines for Creating Tables

When you create an application, you must consider several important factors:

- The characteristics of the database on which you are building.
- The database contents. If the necessary tables in the database do not yet exist, you must design and create them.
- The database design. Does it meet the requirements of your application and other uses for the database?

The Sample Database

In the sample Sales Entry Application, you are working within a database called accounts. For this application, you must store three kinds of information:

- Information about a particular customer: name, customer number, and address.
- Information about a particular order: the type and quantity of the product(s), name and number of the customer, and date ordered.
- Information about the marketing of a particular product: the type of product and quantity ordered.

You can most conveniently store the three kinds of data in three separate tables. Then, making use of a relational database management system's flexibility, you can add data to and retrieve data from one or more of the tables.

The following table shows how information about the three objects—customers, orders and products—is organized in three tables: Customer, Orders, and Marketing.

Table Name	Column Name	Data Type
Customer	Name	char(20)
	Number	smallint
	Address	char(60)
Orders	Product	char(20)
	Quantity	smallint
	Custname	char(20)
	Custnum	smallint
	Current_date	date
Marketing	Prod	char(20)
	Quantity	smallint

The customer number is a unique identifier and can be used to link the tables on customers and orders. You can also use the product name as a link between the tables on orders and marketing.

If these three tables do not already exist in the accounts database, set them up using the Tables utility. To reach this utility, choose Utilities. From the resulting menu, choose Ingres/Menu. From this menu, choose Tables.

If you need help with the Tables utility, see *Character-based Querying and Reporting Tools User Guide.*

Creating a User-Specified Frame

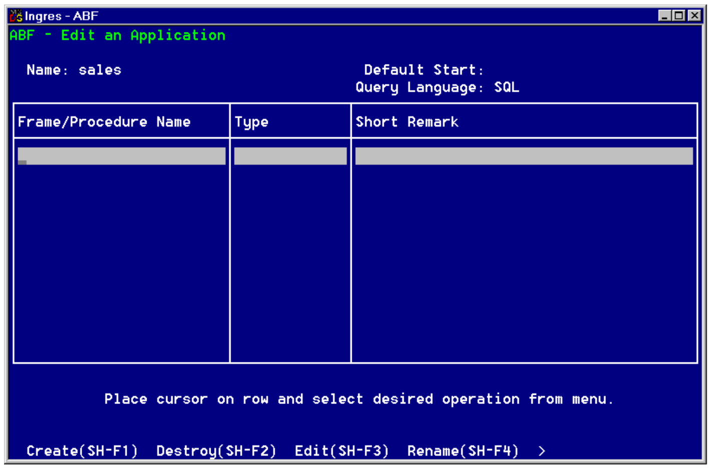
At this point, you have created the Sales Entry Application and the tables in the accounts database. You are ready to define frames and source files. You build each of these application components in three steps:

1. Select the type of the component.
2. Create the component itself.
3. Specify the component in detail in the appropriate definition frame.

These operations begin with the Edit an Application frame, shown in the following figure.

- If you have followed the steps in the previous section, this frame is displayed.
- To display this frame, from the Application Catalog frame, highlight the row for the Sales Entry Application and select Edit.

The first frame to define is the top entry frame of the application.



```
Ingres - ABF
ABF - Edit an Application

  Name: sales                                       Default Start:
                                                   Query Language: SQL

 Frame/Procedure Name | Type  | Short Remark

           Place cursor on row and select desired operation from menu.

  Create(SH-F1)  Destroy(SH-F2)  Edit(SH-F3)  Rename(SH-F4)  >
```

This section covers the following procedures:

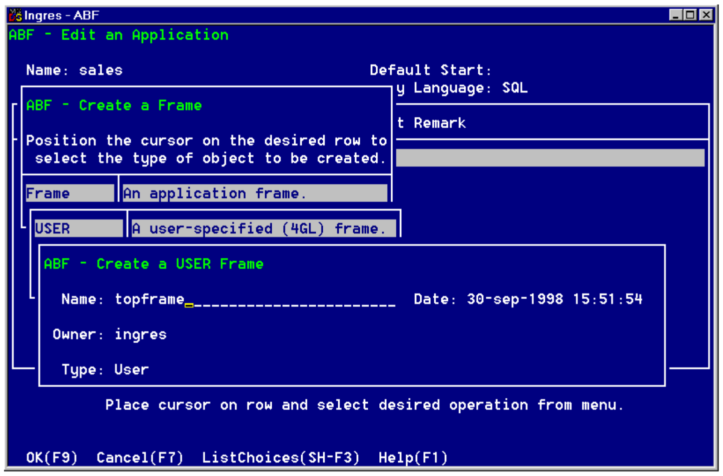
- Define the User-specified frame Topframe
- Create application constants for menu names
- Create the global variable *arr_count*

Create the User Frame Topframe

The first frame in this sample application is a main menu frame called Topframe, which allows users to choose among the various capabilities of the application.

To create the Topframe user frame

1. At the Edit an Application frame, choose Create.
2. When the pop-up frame Create a Frame or Procedure appears, select Frame.
3. When the pop-up frame Create a Frame appears, select USER to display the Create a USER Frame pop-up frame.
4. In the Name field, enter **topframe**. The remaining fields are display only. The frame appears as illustrated in the following figure:

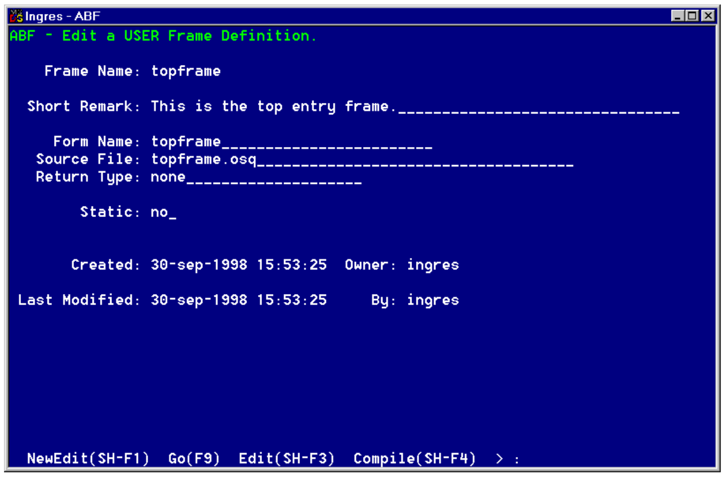


```
ABF - Edit an Application

  Name: sales                                  Default Start:
  ABF - Create a Frame                        y Language: SQL

  Position the cursor on the desired row to   t Remark
   select the type of object to be created.

  Frame      An application frame.
   USER       A user-specified (4GL) frame.
    ABF - Create a USER Frame

      Name: topframe_________________________  Date: 30-sep-1998 15:51:54

     Owner: ingres

      Type: User

            Place cursor on row and select desired operation from menu.

  OK(F9)  Cancel(F7)  ListChoices(SH-F3)  Help(F1)
```

5. To create Topframe, choose OK. ABF creates the frame and adds it to the catalog.

Defining Topframe

The Edit a USER Frame Definition frame appears as shown in the following figure. You are now ready to define the properties of Topframe.



```
Ingres - ABF
ABF - Edit a USER Frame Definition.

    Frame Name: topframe

  Short Remark: This is the top entry frame._______________________________

     Form Name: topframe________________________
   Source File: topframe.osq____________________________________
   Return Type: none____________________

        Static: no_


       Created: 30-sep-1998 15:53:25  Owner: ingres

 Last Modified: 30-sep-1998 15:53:25     By: ingres





  NewEdit(SH-F1)  Go(F9)  Edit(SH-F3)  Compile(SH-F4)  > :
```

ABF places the values you entered in the Create a USER Frame pop-up in the fields of this frame.

- Topframe is the Frame Name.
- The Short Remark field is initially empty. You can enter a brief comment, such as: This is the top entry frame.
- The Form Name field contains the name of the form to use with this frame. The default is the name of the frame. You can change this field.
- The Source File field shows the default name of the file to contain the 4GL code for the frame. This is the name of the frame with the SQL file extension .osq. (The QUEL extension is .osl.) You can change it if you like.
- The Return Type field specifies the data type of the value that the return statement in this frame returns to the calling frame. This can be none (the default), or any Ingres data type.
- Use the Static field to specify whether this frame's data can be saved between calls to the frame (the default is no). See Creating a User-Specified Frame (see page 462).
- The remaining fields give creation and update information about the frame.

Build the Menu for Topframe

The user frame Topframe includes both a form and a menu.

To build the menu for Topframe

1. Choose the Edit operation. You enter the system text editor under the file name *topframe.osq*.
2. You can now enter the 4GL source code needed for the frame. Enter the 4GL code in The 4GL Code for Topframe section.
3. When you exit from the editor, you return to the Edit a USER Frame Definition frame.

Enter the 4GL Code for Topframe

This is the 4GL code for Topframe.

To enter the code

1. Select Edit from the Edit a USER Frame Definition frame to open the file topframe.osq.
2. Enter this code:

```
initialize =
begin
    arr_count := 0;
    /* Global variable arr_count will        */
    /* contain the number of elements in     */
    /* the global array cust_array   */
end

/* Use global constants for menu names */
 sales_opt =
begin
    callframe salerep;
end
 custom_opt =
begin
    callframe customer;
end

 new_opt =
begin
    callframe neworder;
end
 end_opt =
begin
    return;
end
```

This code specifies four operations, set up as global constants: Sales, Customer, New and End. Each operation except End contains a single callframe statement.

End causes the application to exit, because Topframe is not called from any other frame. End contains a return statement that returns you to the ABF Topframe.

The other menu items call frames in the application. You can specify the menu options in the source file even though you have not yet defined the frames. For example, Sales calls the frame Salerep.

This frame uses two types of global application components, a global variable and a set of application constants:

Application Constants

Note the format of the menu item names, for example, :sales_opt. These are application constants you set up using the Globals submenu. See Create Constants (see page 466).

Global Variables

Arr_count is a global variable that acts as a counter for a dynamic array used by the NewOrder frame in this application. The initialize statement at the beginning of the file sets its value to 1. See Create a Global Variable (see page 468).

Create Constants

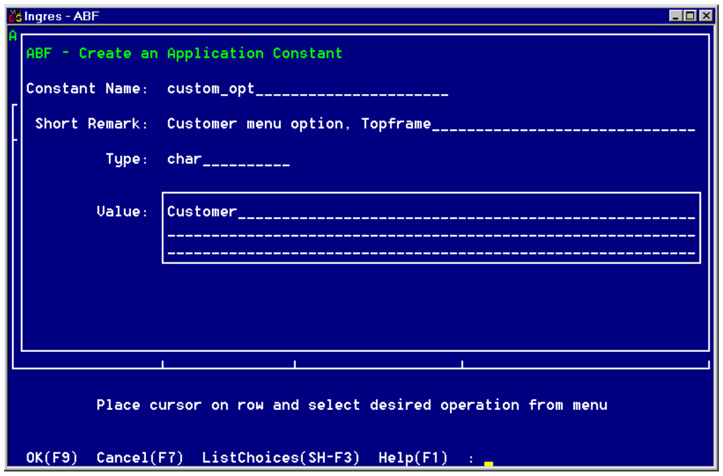
Topframe's menu operation names are set up as application constants.

To create these constants

1. From the Edit a USER Frame Definition frame, select End to exit to the Edit an Application frame.
2. Select the Globals operation to display the Create or Edit Global Components pop-up menu.
3. Highlight Constants and choose Select.

   The Edit Application Constants frame is displayed.
4. Select Create to display the Create an Application Constant pop-up, shown in the following figure.
5. In the Name field, enter **custom_opt**.
6. In the Short Remark field, optionally enter a brief description, such as Customer menu option, Topframe.

7. In the Type field, enter **char**.



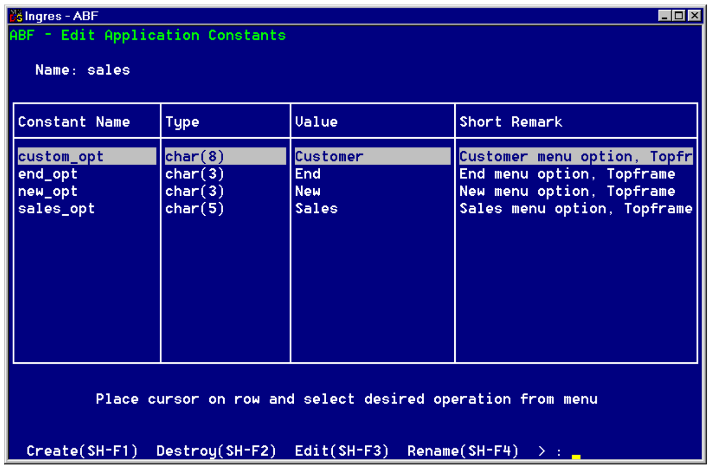
```
ABF - Create an Application Constant

Constant Name:  custom_opt______________________

 Short Remark:  Customer menu option, Topframe______________________________

         Type:  char__________

        Value:  Customer____________________________________________________
                ____________________________________________________________
                ____________________________________________________________

        Place cursor on row and select desired operation from menu

OK(F9)  Cancel(F7)  ListChoices(SH-F3)  Help(F1)  :
```

8. In the Value field, enter **Customer**. This is the menu name that appears on the menu line.
9. Select OK to create the constant.
10. The Edit Application Constants frame appears, displaying the new constant and its features in the table field.
11. Repeat this procedure for each of the remaining three application constants for this frame. Their values are:

Name	**Type**	**Value**	**Short Remark**
End_opt	char	End	End menu option, Topframe
New_opt	char	New	New menu option, Topframe
sales_opt	char	Sales	Sales menu option, Topframe

12. You have now created the application constants used in the Topframe menu. The Edit Application Constants frame displays the constants in its table field, as shown in the following figure. Select End to return to the Edit an Application frame.

For more information about creating application constants, see Building Applications (see page 351).



```
Ingres - ABF
ABF - Edit Application Constants

   Name: sales

 Constant Name    Type           Value              Short Remark

 custom_opt       char(8)        Customer           Customer menu option, Topfr
 end_opt          char(3)        End                End menu option, Topframe
 new_opt          char(3)        New                New menu option, Topframe
 sales_opt        char(5)        Sales              Sales menu option, Topframe

          Place cursor on row and select desired operation from menu

  Create(SH-F1)  Destroy(SH-F2)  Edit(SH-F3)  Rename(SH-F4)  > :
```

Create a Global Variable

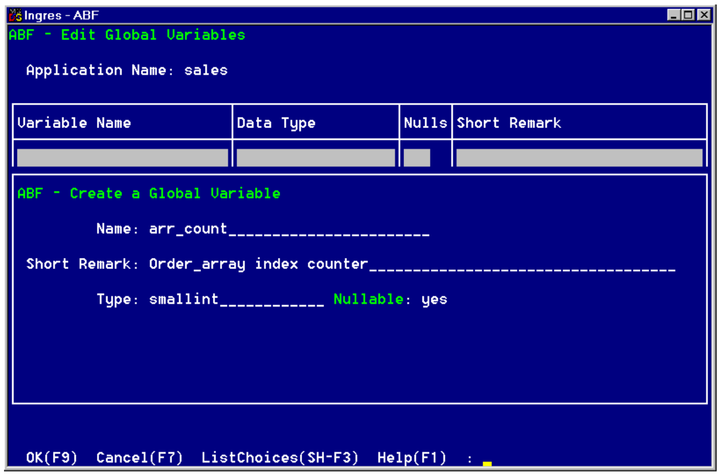
Topframe contains a global variable, *arr_count*. This variable is an index counter for the dynamic array used in another Sales Entry frame, NewOrder. Topframe initializes *arr_count* to a value of "0."

To create this global variable

1. Display the Edit an Application frame.
2. From the Edit an Application frame, select the Globals operation to display the Create or Edit Global Components pop-up menu.
3. Highlight Variables and choose Select.
4. You see the Edit Global Variables frame. Select Create to display the Create a Global Variable pop-up.
5. In the Name field, enter **arr_count**.
6. In the Type field, enter **smallint**. Use the ListChoices menu operation if you want to select from a pop-up menu of data types.
7. In the Nullable field, take the default yes.
8. In the Short Remark field, optionally enter a brief description, such as Order array index counter.

9. The frame appears as in the following figure. Select OK to create the variable.

10. The Edit Global Variables frame appears. The global variable *arr_count* is displayed in the table field. Select End to return to the Edit an Application frame.



```
ABF - Edit Global Variables

  Application Name: sales

 Variable Name        | Data Type        |Nulls|Short Remark

 ABF - Create a Global Variable

          Name: arr_count_______________________

  Short Remark: Order_array index counter___________________________________

          Type: smallint____________ Nullable: yes


  OK(F9)  Cancel(F7)  ListChoices(SH-F3)  Help(F1)  :
```

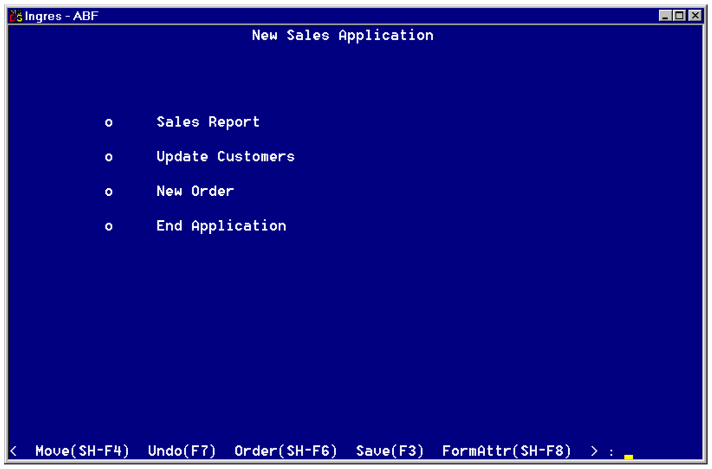
Topframe is almost complete, but you still must define the form. The FormEdit menu operation of the Edit a USER Frame Definition frame gives you direct access to VIFRED so that you can define the form's layout.

To create the form

1. Move to the Edit a USER Frame Definition frame for Topframe.

2. Choose FormEdit. Because you have not changed the default name of the form, VIFRED begins editing Topframe. The VIFRED Creating a Form pop-up tells you that the form does not exist, and offers a menu of form types.

3. Choose the Blank operation from the submenu and use VIFRED operations to create a form similar to the following. The text on this frame is composed completely of trim elements.



```
Ingres - ABF
                         New Sales Application



          o     Sales Report

          o     Update Customers

          o     New Order

          o     End Application









< Move(SH-F4)  Undo(F7)  Order(SH-F6)  Save(F3)  FormAttr(SH-F8)  > :
```

   You have created a menu frame with four elements, not the five that Topframe has when the application is complete.

4. Select End. Save your changes in VIFRED, which returns you to ABF.

For more information, see the description of VIFRED in the *Character-based Querying and Reporting Tools User Guide.*

Check the 4GL Syntax

Now that you have created the form, you can check the syntax of the 4GL code.

The Compile operation in the Edit a USER Frame Definition frame compiles the 4GL code and performs all necessary syntax checking. The Compile operation is optional. If you do not select Compile to perform a syntax check, the 4GL code is automatically compiled when the application is run.

To perform a syntax check

1. Choose the Compile operation.
2. If there are errors or warnings, ABF tells you about them, and offers you the option of displaying and correcting them on the Error Listing Frame. For more information, see Create the 4GL Source Code File for a User-Specified Frame (see page 373).

   If there are no errors, and the code is syntactically correct, you return to the Edit a USER Frame Definition frame.

Add a Long Remark

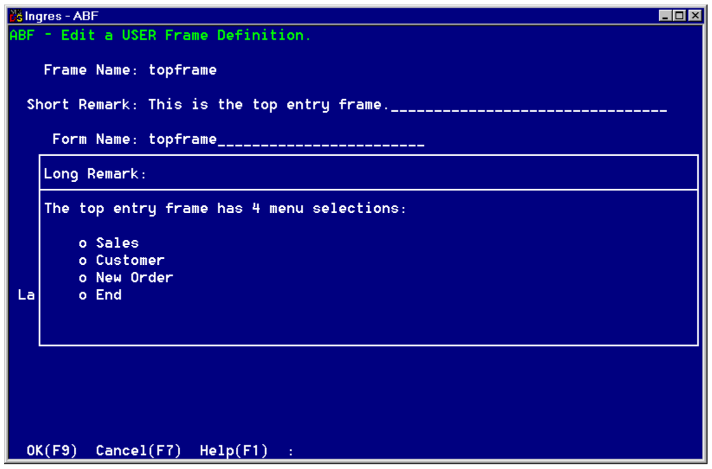
You can optionally add a long remark to describe the frame.

To add a long remark

1. From the Edit a USER Frame Definition menu, choose LongRemark. The LongRemark pop-up frame appears.
2. In the Long Remark field, enter a comment, such as:

```
The top entry frame has 4 menu selections:
-   Sales
-   Customer
-   New Order
-   End
```

The window appears as in the following figure:



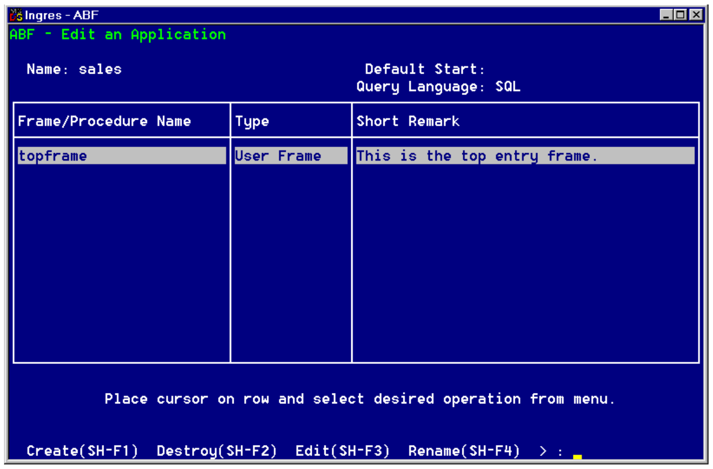
```
ABF - Edit a USER Frame Definition.

    Frame Name: topframe

  Short Remark: This is the top entry frame.________________________________

     Form Name: topframe________________________
    Long Remark:

    The top entry frame has 4 menu selections:

        o Sales
        o Customer
        o New Order
 La     o End



  OK(F9)  Cancel(F7)  Help(F1)  :
```

3. Choose OK to return to the Edit a USER Frame Definition frame.

Test Topframe

You have now completed the definition of Topframe. Choose End to return to the Edit an Application frame.

You can test the current state of the application to see if it is defined properly. The Edit an Application frame now looks as it does in the following figure. In the Frame/Procedure Name table field, you see an entry for "topframe." The names, types, and Short Remarks for frames appear here as you define them.



```
ABF - Edit an Application

  Name: sales                                 Default Start:
                                             Query Language: SQL
```

Frame/Procedure Name	Type	Short Remark
topframe	User Frame	This is the top entry frame.

```
             Place cursor on row and select desired operation from menu.

  Create(SH-F1)  Destroy(SH-F2)  Edit(SH-F3)  Rename(SH-F4)  > :
```

The Sales Entry application contains only one frame.

To test the application at this point

1. Choose Go. ABF prepares to run the Sales Entry Application. ABF prompts:

   ```
   Start execution at 'Topframe'?
   ```

2. Respond **y** (yes) and press Return. ABF begins executing the application. ABF:

   - Examines the application's definition to see if any frames or files must be compiled
   - Displays status messages about the compiling and linking operations

     For example, when ABF compiles Topframe, the following message appears at the bottom of the window:

     ```
     Compiling 'topframe.osq'...
     ```

   - Displays messages about any errors that it detects, prompts you to correct them, and recompiles the application
   - When finished compiling, forms an interpreted image of the application and executes it

Set Default Start Frame

If you want the application to automatically start running at Topframe each time, you can optionally set this frame up as the default start frame for the application.

To set the default start frame

1. Return to the Edit an Application frame and select the Defaults menu operation.
2. In the Default start frame field, enter **topframe** and select OK.

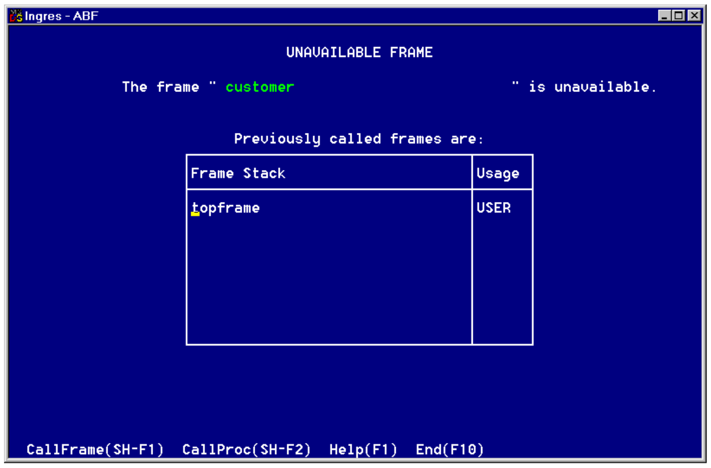
How You Can Test with an Undefined Frame

The application's initial frame now appears. The frame Topframe is now running.

1. You can choose any of the menu operations available in Topframe. Start by choosing Customer.

   The operation Customer calls the frame Customer. Because this frame has not yet been defined, ABF intercepts the call and summons a special frame, shown in the following figure.

   This frame is part of the ABF debugging environment. It appears when you call an undefined frame. The frame's table field lists frames that were called before the current one, the most recent at the top. (In this case, the only frame called was Topframe.)



```
                         UNAVAILABLE FRAME

        The frame " customer                        " is unavailable.

                     Previously called frames are:
```

Frame Stack	Usage
topframe	USER

   Because this frame is part of ABF, you can continue with the application.

2. To test other frames, use the Call menu operation. Call allows you to call other frames directly.

   When you choose Call, the "Frame To Call" prompt appears at the bottom of the window. If other frames were defined for the application, call them by entering the frame name and pressing Return.

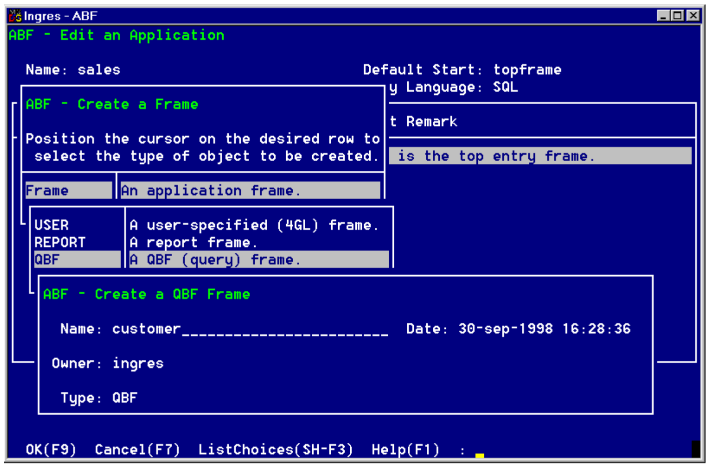
   Because no frames are defined, choose End. ABF returns to Topframe. You exit the application and return to the Edit an Application frame.

Define a QBF Frame

You are now back in the Edit an Application frame, where you can go on to create the rest of the frames for the sample application. Next, create the Customer frame, the QBF frame in the application.

To define a QBF frame

1. Choose Create from the menu.
2. From the Create a Frame or Procedure pop-up, select Frame.
3. Select QBF from the Create a Frame pop-up. The Create a QBF Frame pop-up appears.
4. In the Name field, type customer. The window appears as shown in the following figure:

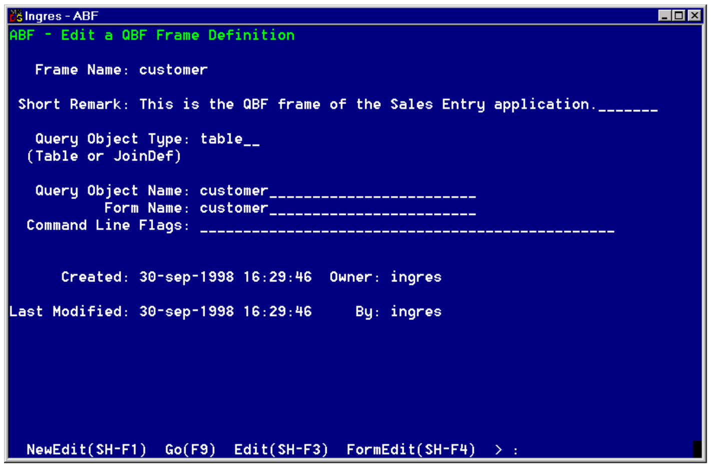


```
Ingres - ABF
ABF - Edit an Application

  Name: sales                                  Default Start: topframe
                                              y Language: SQL
  ABF - Create a Frame
                                              t Remark
  Position the cursor on the desired row to
   select the type of object to be created.   is the top entry frame.

  Frame       An application frame.

   USER       A user-specified (4GL) frame.
   REPORT     A report frame.
   QBF        A QBF (query) frame.

    ABF - Create a QBF Frame

      Name: customer________________________  Date: 30-sep-1998 16:28:36

     Owner: ingres

      Type: QBF

  OK(F9)  Cancel(F7)  ListChoices(SH-F3)  Help(F1)  :
```

5. Choose OK to create the Customer frame. The Edit a QBF Frame Definition frame appears, showing default values for the Customer frame in the Frame Name, Query Object Type, and Query Object Name fields.
6. In the optional Short Remark field, enter a brief description of the frame if you like.
7. In the Query Object Type field, enter **table**.

8. The Query Object Name field shows the frame name (customer) as a default. In the Form Name field, enter **customer**.

The Edit an Application frame now appears as in the following figure:



```
ABF - Edit a QBF Frame Definition

   Frame Name: customer

 Short Remark: This is the QBF frame of the Sales Entry application._______

   Query Object Type: table__
  (Table or JoinDef)

   Query Object Name: customer________________________
           Form Name: customer________________________
  Command Line Flags: ________________________________________________


      Created: 30-sep-1998 16:29:46  Owner: ingres

Last Modified: 30-sep-1998 16:29:46     By: ingres





  NewEdit(SH-F1)  Go(F9)  Edit(SH-F3)  FormEdit(SH-F4)  > :
```

The Command Line Flags field holds any QBF command line arguments. For this application, do not use any flags here.

See Create a QBF Frame (see page 382) for information on command line flags.

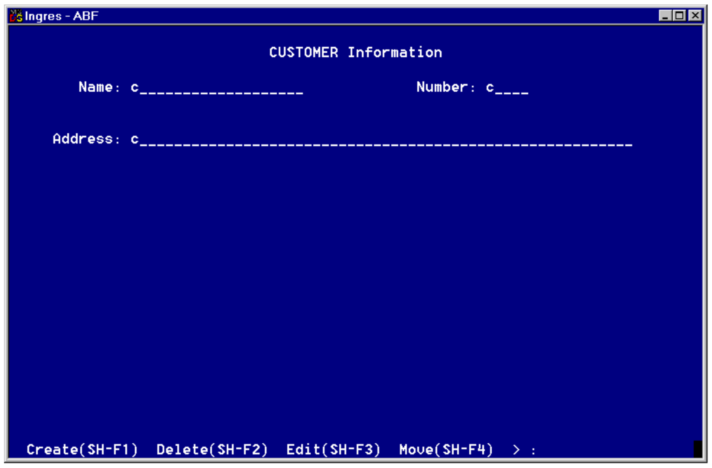
Define the Customer Form

The next step is to specify the form for the QBF Customer frame. In the last sequence of steps, you entered Customer in the Form Name field.

To define the customer form

1. Choose the FormEdit operation to invoke VIFRED on the form. You see a message telling you that the frame is being saved. You can construct a form for the QBF frame in the same way you did for the user-specified frame Topframe. In this case, however, the form is based on the Customer table.
2. When you see the VIFRED Creating a Form frame, highlight the Table frame type and enter **Select**.

3. At the Creating a Form Based on a Table frame, enter the name **customer** and the Display Format simplefields. Select OK.
4. The fields you see are columns from the Customer table. Starting with these, define a form similar to the one in the following figure. Be sure to use the name Customer in the title. Save the form and exit from VIFRED.



```
                         CUSTOMER Information

      Name: c__________________              Number: c____

   Address: c______________________________________________________

Create(SH-F1)  Delete(SH-F2)  Edit(SH-F3)  Move(SH-F4)  > :
```

You do not enter a QBF Name while in VIFRED because ABF automatically associates the Customer database table with the Customer form based on the values you entered in the Edit a QBF Frame Definition frame.

Test the Customer Frame

After completing the specifications for the Customer frame, test the frame by running it directly from the Edit a QBF Frame Definition frame.

To test the customer frame

1. Choose Go. Because Customer is a QBF frame, QBF is invoked on the table and form you specified.
2. You did not use flags on the Edit a QBF Frame definition frame. This means that, at the QBF Execution Phase frame menu, you must select the mode in which you want to run this frame. For example, select Update.
3. You see the Customer Information form with the QBF menu line, as shown in the illustration of the Customer Information frame. You can proceed to use QBF.
4. Call the Customer frame from the QBF Execution Phase frame menu in different QBF modes so that you can enter, update, and retrieve the data in the Customer table.

When you exit QBF, the frame Customer ends, and the Edit a QBF Frame Definition frame reappears.

Building a Report Frame

You define the report frame for the application from the Edit an Application frame. This section covers two useful procedures:

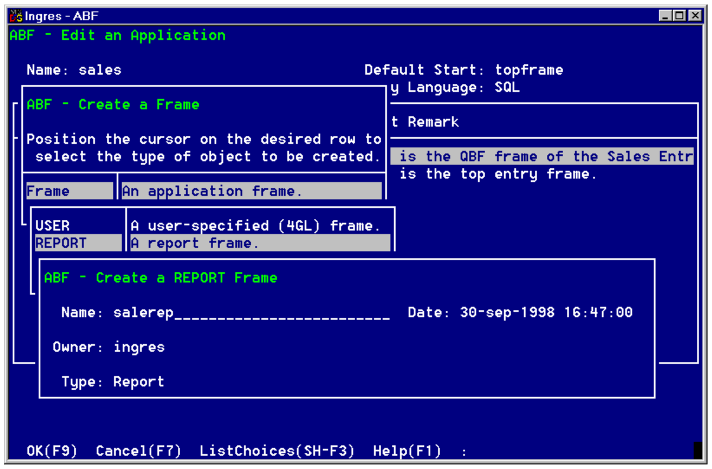
- Define a Report frame
- Create a Pop-up frame

Define the SaleRep Frame

SaleRep is a Report frame that calls the Report-Writer to produce a report on the quantities of a particular product sold. You display the pop-up Salerep frame by selecting Sales from the Topframe menu.

To create this frame

1. Choose Create from the Edit an Application menu.
2. From the Create a Frame or Procedure pop-up, select Frame.
3. The Create a Frame pop-up appears. Select the REPORT frame type.
4. The Create a REPORT Frame appears. In the Name field, type salerep. The following figure shows this stage:



```
Ingres - ABF
ABF - Edit an Application

  Name: sales                               Default Start: topframe
                                            y Language: SQL
  ABF - Create a Frame
                                            t Remark
  Position the cursor on the desired row to
   select the type of object to be created.  is the QBF frame of the Sales Entr
                                             is the top entry frame.
  Frame       An application frame.

   USER        A user-specified (4GL) frame.
   REPORT      A report frame.

    ABF - Create a REPORT Frame

      Name: salerep_________________________  Date: 30-sep-1998 16:47:00

     Owner: ingres

      Type: Report


  OK(F9)  Cancel(F7)  ListChoices(SH-F3)  Help(F1)  :
```

5. Choose OK to load a default report definition for Salerep into the Report layout frame. The Edit a REPORT Frame Definition frame appears, shown in the following figure.
6. The Frame Name is salerep. If you like, enter a brief description in the Short Remark field.
7. In RBF Report (y/n) field, enter **n**. Salerep calls Report-Writer and thus uses a report source file.

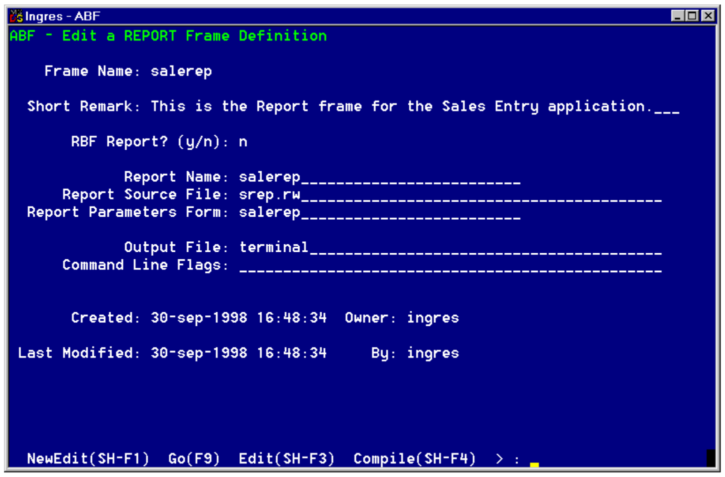
8. In the Report Source File field, enter the name **srep.rw**, because that is the name of the source file that you are entering for Salerep in the next section.

9. In the Report Parameters Form field, enter the name of the form that the user can use to enter the parameter or variable. This can be the same as the frame name and the report name.

10. The Output File field indicates the type of output for the report. This can be Printer, Terminal, or a file name. Enter **terminal** here.

    The frame also contains a Command Line Flags field in which you can enter command line arguments for running reports. For more information on Command Line Flags, see Creating Report Frames (see page 375).

The following figure shows the completed frame. The next section describes the Report-Writer source file srep.rw.



```
ABF - Edit a REPORT Frame Definition

    Frame Name: salerep

  Short Remark: This is the Report frame for the Sales Entry application.___

       RBF Report? (y/n): n

             Report Name: salerep_________________________
      Report Source File: srep.rw_________________________________________
  Report Parameters Form: salerep_________________________

             Output File: terminal_______________________________________
      Command Line Flags: ________________________________________________


       Created: 30-sep-1998 16:48:34  Owner: ingres

 Last Modified: 30-sep-1998 16:48:34     By: ingres



  NewEdit(SH-F1)  Go(F9)  Edit(SH-F3)  Compile(SH-F4)  > :
```

Enter a Report-Writer Specification Using sreport

In the Edit a Report Frame Definition frame, the Report Source File field names a source file that contains the Report-Writer commands for the Salerep report. If you select RBF to define the report, the Report Source File field remains blank.

The example in this chapter uses the source file srep.rw. You already entered this name in the Report Source File field in the previous section.

To enter the Report-Writer specification for the example into this file

1. Select Edit.

   The report is named Salerep and contains one variable name, *product_value*. The user defines the variable (in this case, the type of product reported on) in the query that retrieves the data.

```
.name salerep
.query
   select product, custname, quantity
     from orders
     where product = '$product_value'
.declare product_value = varchar(20)
 with prompt "Enter Product: "
.position product (0,30),
 custname (35,50), quantity (70,4)
.sort product, custname

.head report
   .underline
     .center
     .print 'Sales Report'
   .nounderline
   .nl 2
.head product
   .underline
     .left product    .pr 'Product Name'
     .left custname   .pr 'Customer Name'
     .left quantity   .pr 'Quantity'
   .nounderline
   .nl
   .tformat product(c30)

.head custname
   .nl
   .tformat custname(c30)
.detail
   .tab product                 .pr product (b30)
   .tab custname                .pr custname (b30)
   .tab quantity                .pr quantity (f4)
   .nl
```

2. Save the text file. When you leave your editor, ABF returns you to the Edit a REPORT Frame Definition frame.

3. Choose Compile. This invokes sreport. The report name has the same name as the frame, so salerep appears in the Report Name field.

After the source file is compiled, you return to the Edit a REPORT Frame Definition frame.

Define the SaleRep Pop-up Form

The Report Form field contains the name of a form in which the user can enter parameters for the report. This makes entering parameters easy and provides a consistent interface for the user.

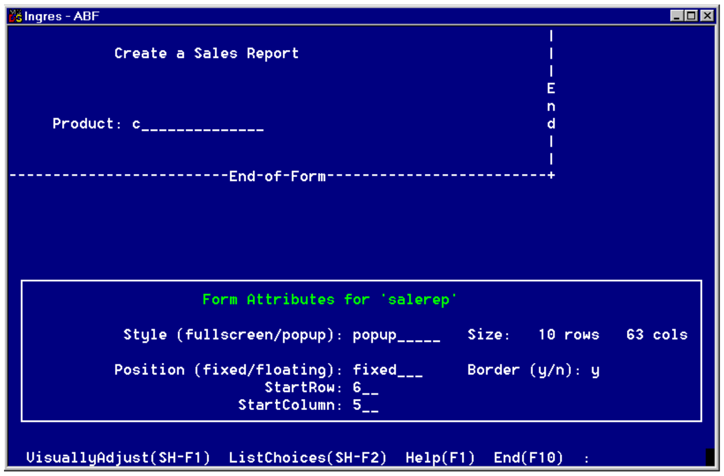
A form is optional in a report frame. If you do not enter one, ABF calls the Report-Writer directly when the user runs the report frame. The FormEdit operation is not active unless you specify a form.

The Salerep frame has a pop-up form. Creating a pop-up is the same as creating a fullscreen form, with the exceptions noted in the steps below.

The report has one variable, *product_value*, which the user specifies. You need a form that has a single field with the same name as the variable. In this case, the form requires a field with the name *product_value*. The "$" is not part of the prompt name—do not include the "$" in the field name.

To define the pop-up form

1. Choose the FormEdit operation to call VIFRED. Choose the BLANK option at the VIFRED Creating a Form Menu.
2. Create a form similar to the one in the following figure. Because this is a pop-up, use the VIFRED FormAttr option to change the style to pop-up. The figure below shows this stage. The Salerep pop-up is positioned near its associated menu listing.
3. Choose VisuallyAdjust. The Resize option allows you to move the cursor to change the size of the form and determine its displayed position on top of Topframe. Exit back to ABF.



```
            Create a Sales Report                          |
                                                           |
                                                           |
                                                           E
                                                           n
     Product: c______________                              d
                                                           |
                                                           |
-------------------------End-of-Form-----------------------+



                    Form Attributes for 'salerep'

           Style (fullscreen/popup): popup_____   Size:   10 rows   63 cols

          Position (fixed/floating): fixed___     Border (y/n): y
                           StartRow: 6__
                        StartColumn: 5__

  VisuallyAdjust(SH-F1)  ListChoices(SH-F2)  Help(F1)  End(F10)  :
```

Test the Report Frame

After you build the frame, try testing it from the Edit a REPORT Frame Definition frame. Select **Go** to compile and run the report frame on its own. The frame for SaleRep is displayed, as shown in the illustration of the Sales Entry Application frame.

Alternatively, you can check to see how the entire application is shaping up in the following way:

1. Use the Compile operation on the report source file.
2. Choose End to return to the Edit an Application frame.
3. Highlight topframe.
4. Choose Go. If you did not set up a default start frame, ABF prompts: "Start execution at 'topframe'?" Enter **y** (yes).
5. The application is now running.
6. To test the report frame that you just defined, choose Sales. Topframe now calls SaleRep.

Run the SaleRep Frame

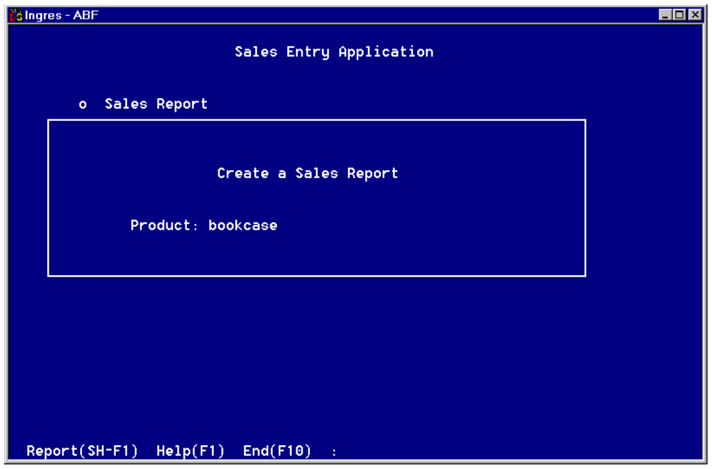
Display the SaleRep frame by following the steps in the previous section. The menu in the window is the one ABF automatically supplies for all Report frames:

```
Report Help End
```

The SaleRep form contains a single field displaying the name of a product. Assume that there is data in the orders table.

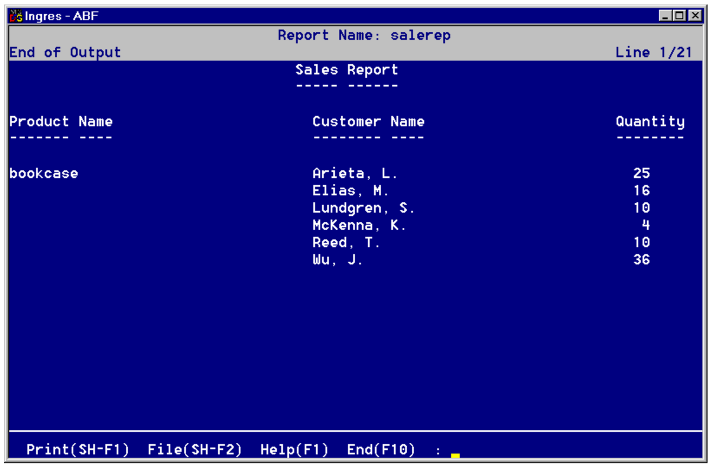
To run the frame

1. Enter the product bookcase, which results in the window shown in the following figure:



```
Ingres - ABF
                         Sales Entry Application

      o  Sales Report

                     Create a Sales Report

              Product: bookcase

 Report(SH-F1)  Help(F1)  End(F10)  :
```

2. Choose the Report operation. The Report-Writer runs, and the output appears on your screen. Sample results are shown in the following figure:



```
Ingres - ABF
                              Report Name: salerep
End of Output                                                     Line 1/21
                                Sales Report
                                ----- ------

Product Name                       Customer Name                  Quantity
------- ----                       -------- ----                  --------

bookcase                           Arieta, L.                        25
                                   Elias, M.                         16
                                   Lundgren, S.                      10
                                   McKenna, K.                        4
                                   Reed, T.                          10
                                   Wu, J.                            36

 Print(SH-F1)  File(SH-F2)  Help(F1)  End(F10)  :
```

3. When the display is finished, press Return to clear the screen. The frame SaleRep is redisplayed so that you can run the report again.

4. When you have finished generating reports, choose End, which returns you to Topframe.

5. From Topframe, choose End to exit the application. You return to the Edit an Application frame.

Another User-Specified Frame

The frame NewOrder (see the NewOrder Frame section) allows you to enter data into the Sales Entry Application tables so that you can test the Report frames.

NewOrder uses an array based on a record type. This section contains directions for the following procedures:

- Define the NewOrder frame
- Create the Record Type *cust_record*
- Create the Dynamic Array *cust_array*

Define the NewOrder Frame

To start creating the NewOrder frame

1. Choose the operations to define a user-specified frame named neworder. ABF displays the Edit a USER Frame Definition frame.
2. Fill in the fields as you did with Topframe. Take the default values where possible.
3. At this point you can edit the 4GL file for the frame. Choose Edit and enter the source code shown below.

The NewOrder frame places data in two tables, "Orders" and "Marketing." The 4GL code for NewOrder is:

```
initialize =
declare
        cnum = integer;
        rcount = smallint;
        h_count = smallint;
        hret = integer;
 begin
end
```

```
'Add' =
begin
        if custnum = 0 then
                callproc beep(); /* Built-in procedure */
                message 'You must enter a customer' +
                        ' number before selecting ADD';
                sleep 2;
                resume;
        endif;
        insert into orders (product, custname,
                custnum, quantity, current_date)
                values (:product, :custname, :custnum,
                        :quantity, 'today');
        commit;
        insert into marketing (prod, quantity)
                values (:product, :quantity);
        commit;

        /* Check to see if a record exists in */
        /* the table Customer having the */
        /* customer number in field Custnum */
        neworder = select cnum = number
        from customer
                where number = :custnum;
        /* See if a record already exists */
        inquire_sql (rcount = rowcount);
        if rcount = 0 then
                insert into customer (name, number, address)
                        values (:custname, :custnum,
                                'Whereabouts unknown');
                commit;
        endif;

        /* Look at each record of the array              */
        /* cust_array, until an entry is found   */
        /* for the current customer number               */
        h_count := 1;
        while (h_count <= arr_count) do
                if cust_array[h_count].c_number =
                        :custnum then
                        endloop;
                endif;
                h_count := h_count + 1;
        endwhile;
        /* If no entry is found, add one,                */
        /* otherwise give a warning message      */
        if (h_count > arr_count) then
                arr_count := arr_count + 1;
                cust_array[arr_count].c_name := :custname;
                cust_array[arr_count].c_number :=
                        :custnum;
        else
                message 'WARNING: You have already' +
                        ' entered an order for ' +
                         varchar(:custname) +
                        ' during this session'
                         with style = popup;
        endif;
        clear field all;
 end

'Find' =
begin
        if custnum = 0 then
                message 'You must enter a customer' +
                        ' number before selecting FIND';
                sleep 2;
                resume;
        endif;
/* Write the customer name that                  */
/* corresponds to the customer number    */
/* in the Custnum field into the                         */
/* Custname field.  */
        neworder = select custname = name
                from customer
                where number = :custnum;

/* See if a row exists for that   */
/* customer number */
        inquire_sql (rcount = rowcount);
        if rcount = 0 then
                message 'No such customer number';
                sleep 2;
                resume;
        endif;
        clear field product, quantity;
 end

'ListChoices' =
begin
        /* List choices for field.  For the      */
        /* product field this displays                   */
        /* possible products, taken from the     */
        /* Validation Check To Perform                   */
        /* attribute for this field.     */
        hret = callproc help_field();
                /* Built-in procedure */
end
'End', key frskey3 =
begin
        return;
 end
```

The code declares a menu with four operations: Add, Find, ListChoices, and End. The first two operations contain examples of query statements (insert and select).

ListChoices calls the help_field() built-in procedure. When called for the Product field, this option displays a pop-up list of products from which the user can choose. To specify the product list, use the Validation Check To Perform field in the VIFRED Creating a Form frame while creating the form. See the 4GL part of this guide for more information about built-in procedures.

Arr_count, the global variable you set up while creating Topframe, is used in NewOrder. *Arr_count* gives the number of rows in the dynamic array *cust_array*.

NewOrders references the array *cust_array*, based on the record type *cust_record*. The customer name and number in the selected row, contained in *cust_record*, are inserted into *cust_array*, as element number "*arr_count*" of the array. You can use an array of this type in many ways in an application. NewOrders uses the array to check for duplicate customer entries and displays a warning message if the user enters a duplicate customer. The next two sections show how to create these two global components.

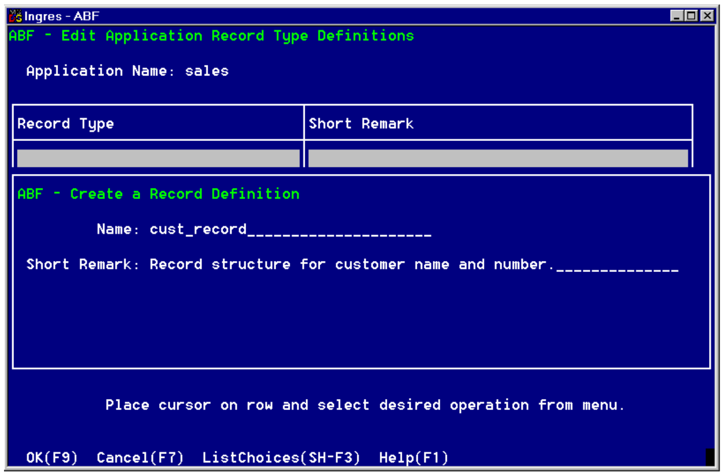
Create a Record Type

In NewOrder, the array *cust_array* is based on the record type *cust_record*. *Cust_record* has as its attributes the simple data types, *c_name* (char) and *c_number* (integer).

To create this record type

1. From the Edit a USER Frame Definition frame, select End to exit to the Edit an Application frame.
2. Select the Globals operation to display the Create or Edit Global Components pop-up menu.
3. Highlight RecordTypes and choose Select.

4. You see the Edit Application Record Type Definitions frame. Select Create to display the Create a Record Definition pop-up.



```
Ingres - ABF
ABF - Edit Application Record Type Definitions

  Application Name: sales

Record Type                     | Short Remark

ABF - Create a Record Definition

        Name: cust_record_____________________

 Short Remark: Record structure for customer name and number.______________

         Place cursor on row and select desired operation from menu.

  OK(F9)  Cancel(F7)  ListChoices(SH-F3)  Help(F1)
```

5. In the Name field, enter **cust_record**.

6. In the Short Remark field, optionally enter a brief description, such as Record structure for customer name and number. This stage is shown in the preceding figure.

7. Select OK to create the record type.

8. The Edit a Record Type Definition frame appears. In its fields are the values you entered in the Create a Record Definition pop-up. *Cust_record* has two attributes, *c_name* and *c_number*. To create the attributes of the record, select Create to display the Create a Record Attribute Frame.

   a. To create c_name, complete this frame. Enter the data type **char(20)** and the optional Short Remark, Customer Name.

   b. Select OK. You return to the Edit a Record Type Definition frame, where the attribute is displayed in the table field.

   c. Repeat the procedure for the attribute c_number, which has a data type smallint and the optional Short Remark Customer number.

9. After creating the attributes, select End to return to the Edit Application Record Type Definitions frame, where the record type you created appears in the table field. Select End.

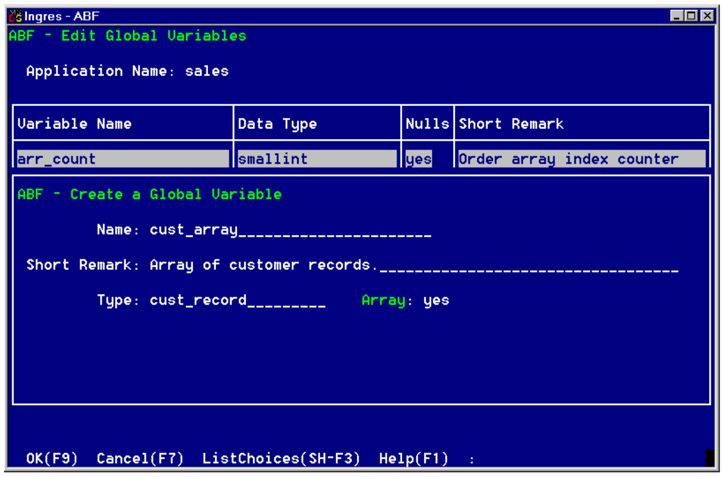
Create a Global Array

NewOrder uses an array that you create as a global variable through the Globals menu. An array is a collection of records that you can treat as a single unit. In 4GL, arrays must be made up of records, not of simple data types. The array in NewOrders is based on the record type *cust_record*, created in Create a Record Type (see page 486).

Cust_array is a global variable. However, you can also use 4GL code to declare an array locally. For information on declaring an array locally, see the 4GL part of this guide.

To create *cust_array*

1. From the Edit an Application frame, select the Globals operation to display the Create or Edit Global Components pop-up menu.
2. Highlight Variables and choose Select.
3. When the Edit Global Variables frame appears, select Create to display the Create a Global Variable pop-up.
4. In the Name field, enter **cust_array**.
5. In the Type field, enter **cust_record**. For an array, you must type this in, as the ListChoices menu of Ingres types does not list user-created record types.
6. When the Nullable field turns into the Array field, enter **yes**.
7. In the Short Remark field, optionally enter a brief description, such as Array of customer records. Use the LongRemark pop-up to enter a longer comment if you like.
8. When the frame appears as in the following figure, select OK to create the variable.


```
Ingres - ABF
ABF - Edit Global Variables

  Application Name: sales

 Variable Name            Data Type          Nulls Short Remark
 arr_count                smallint           yes   Order array index counter

 ABF - Create a Global Variable

         Name: cust_array______________________

  Short Remark: Array of customer records.__________________________________

         Type: cust_record_________     Array: yes



  OK(F9)  Cancel(F7)  ListChoices(SH-F3)  Help(F1)  :
```

9. To return to the Edit Global Variables frame, select End.

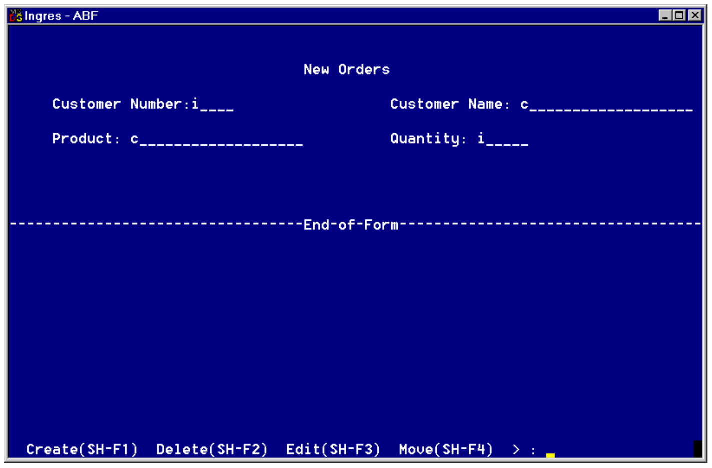
This frame is where this array is listed in the table field.

Define the NewOrder Form

To build the form for NewOrder, redisplay the Edit a USER Frame Definition frame. The NewOrder form is shown in the following figure.

To define the form

1. Enter the form name orders and select FormEdit. At the VIFRED Creating a Form menu, select Table.
2. At the Creating a Form Based on a Table pop-up, specify the Orders table and the field type simplefields. Select OK.



```
                                 New Orders

     Customer Number:i____                   Customer Name: c___________________

     Product: c___________________           Quantity: i_____



--------------------------------End-of-Form----------------------------------



  Create(SH-F1)  Delete(SH-F2)  Edit(SH-F3)  Move(SH-F4)  > :
```

3. The default form includes the fields Custnum, Custname, Product and Quantity.

   Use VIFRED operations to give the fields different titles and reposition them, as shown in the preceding figure. Remove the *current_date* field, or give it the Invisible attribute. Notice that the Internal Name of each field is the same as the corresponding Customer table column.

4. To add a list of products to be displayed when the user selects ListChoices for the Product field, enter the product list in the Validation Check to Perform field in VIFRED. Use this format:

   ```
   product in ["bookcase", "chair", "lamp", "sofa",
   "stool", "table", "lawn mower"]
   ```

5. Save and exit. You return to the Define a USER Definition frame.
6. Choose End to compile the NewOrder frame and return to the Edit an Application frame. The Error Listing frame appears if the source code file contains errors.

After the form is built, you can test the frame.

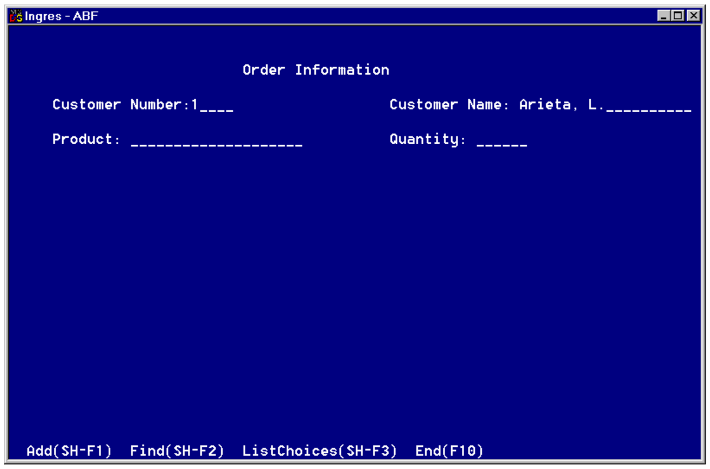
Test the Application

After you define the 4GL source code and form, you can test the newly completed application by trying to run it.

To test the application

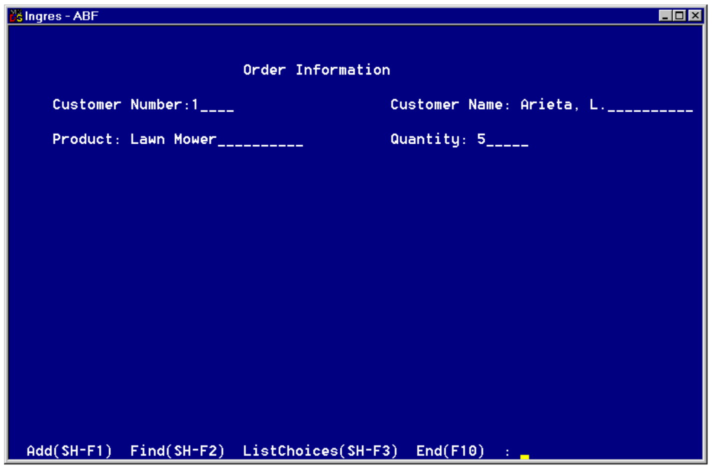
1. If necessary, choose End from your current window to return to the Edit an Application frame.
2. Highlight the frame from which you want the application to start.
3. Choose Go.
4. If you highlighted Topframe, the resulting prompt reads: "Start execution with 'Topframe'?" Enter **y** and press Return to run the application.

   If you have set up a default start frame, you see a pop-up menu asking if you want to start from the highlighted frame or the default. Select the frame you want.
5. Choose New from the Topframe to try the Find and Add operations. The NewOrder frame appears, as shown in the illustration of the New Orders frame.
6. For example, enter **1** in the customer number field and choose Find. Find examines the database for any customer with the number 1 and displays the information in the form. For example, if the customer "Arieta, L." has the number 1, the form appears as in the following figure:



```
Order Information

Customer Number:1____                Customer Name: Arieta, L.__________

Product: ____________________        Quantity: ______
```

7. Enter the product name Lawn Mower and the quantity 5. Select ListChoices to see a pop-up list of products from which you can choose. The form now appears as in the following figure:



```
                    Order Information

   Customer Number:1____              Customer Name: Arieta, L.__________

   Product: Lawn Mower__________      Quantity: 5_____
```

```
 Add(SH-F1)  Find(SH-F2)  ListChoices(SH-F3)  End(F10)  :
```

8. Choose Add. This adds the data to the database.
9. Check the success of this operation by returning to Topframe and choosing the Sales operation to display the report for Lawn Mower. The new report reflects the changes in the database.

With the definition of these two operations, your sample application is now complete.

Installing the User Application

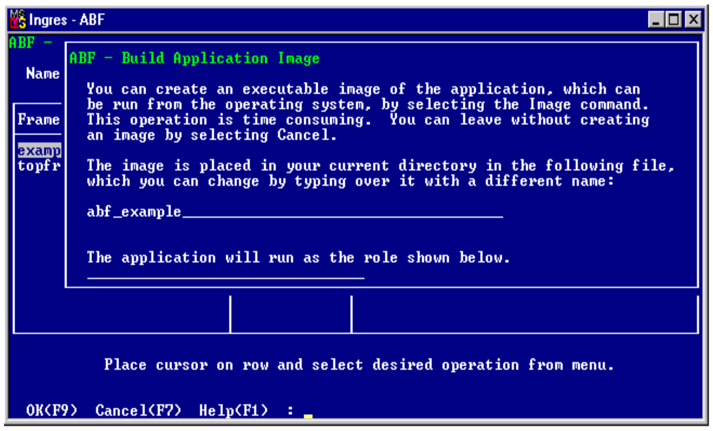
The Sales Entry Application is now complete. You can use Go to run the application from within ABF to ensure that it operates correctly. You are not quite finished from the user's point of view, however. To allow users to run the application from the operating system, use the Image operation from the Utilities menu to create an executable image of your application. You can then specify one or more symbols to start the application. For more information on this process, see Creating an Image of the Application (see page 429).

Create an Executable Image

To build an executable image of the Sales Entry Application, you must start at either the Applications Catalog or the Edit an Application frame.

To create an executable image

1. Choose the Utilities operation.
2. Choose the Image operation. The Build Application Image pop-up frame appears, as shown in the following figure:



```
Ingres - ABF
ABF -
  Name
Frame
examp
topfr

ABF - Build Application Image

  You can create an executable image of the application, which can
  be run from the operating system, by selecting the Image command.
  This operation is time consuming.  You can leave without creating
  an image by selecting Cancel.

  The image is placed in your current directory in the following file,
  which you can change by typing over it with a different name:

  abf_example_____________________________________

  The application will run as the role shown below.
  ________________________________

          Place cursor on row and select desired operation from menu.

  OK(F9)  Cancel(F7)  Help(F1)  : _
```

   The default file name for the executable image is sales. ABF can combine the name of the application with the file extension .exe, depending on your operating system.

   You can change the name of the executable image by entering a new name here. ABF creates the file for the image in your current directory.

3. Take the default for the other fields, if possible.
4. Select the OK operation. ABF builds the executable image. When it is finished, the Utilities menu returns.

If you decide not to create an executable image, return to the Edit an Application frame menu by choosing the Cancel operation.

Application Handling Operations

After you complete an executable image of the Sales Entry Application, you can adapt the application to different uses and types of access. The following statements summarize the operations and their commands. For a more detailed explanation about these procedures, see the corresponding sections in Building Applications (see page 351).

- Run the executable image.
- Define an operating system command to run the application.
- Run the application on a different database.

Run the Image

To run the Sales Entry Application from operating system level, define a name for the application to be used as a command. This example uses the name saleapp:

Windows: Include the following command in a shortcut named saleapp:

```
full_pathname\sales.exe
```

The *full_pathname* is the full pathname of the directory for sales.exe.

UNIX:

```
alias saleapp full_pathname/sales.exe
```

The *full_pathname* is the full pathname of the directory for sales.exe.

VMS:

```
saleapp :== $dir_spec sales.exe
```

The *dir_spec* is the full directory specification of the file sales.exe.

To run the application from the operating system level, type the name, followed by the name of the first frame to call. At this point, the following command runs the Sales Entry Application, starting at the frame Topframe:

```
saleapp topframe
```

You can pass database flags into the application at the operating system prompt. These are described further in your query language reference guide. For example:

```
saleapp -u newuser topframe
saleapp -xf topframe
```

Define an Operating System Command

You can give the application user the convenience of typing in only one name to run the application by creating a command with the frame name as part of it. The following command designates "sales" as the name that users can type in to run the Sales Entry Application and Topframe as the start frame.

Windows:

Put the following command in a shortcut called "sales":

```
full_pathname\sales.exe topframe
```

After this, users need only select the sales shortcut to run the Sales Entry Application.

You can create shortcuts to allow users to start the application on different frames.

UNIX: Define the alias or shell script name:

```
alias sales 'full_pathname/sales.exe topframe'
```

To run the Sales Entry Application at the operating system level, users type sales.

If you defined saleapp as in the previous section, define "sales" as follows:

```
alias sales 'saleapp topframe'
```

VMS: To designate "sales" as the name that users type to run the Sales Entry Application by calling Topframe, define the symbol:

```
sales :== "$dir_spec sales.exe topframe"
```

To run the Sales Application at the operating system level, users type sales.

If you defined saleapp as in the previous section, define "sales" as follows:

```
sales == saleapp + " topframe"
```

You can define aliases that allow users to start the application on different frames. For more information, see Defining Symbols for Different Users (see page 438).

Run the Application on a Different Database

If you created the Sales Entry Application on a *test database* and want to run it on a *production database*, you can do this in either of the ways described below.

In the first method, the Sales Entry Application remains in the test database in which you developed it (Testdb) and created an executable image (*sales.exe*).

To use this application on the Proddb database, make sure that all the components in the application—forms, reports and tables—exist in the production database (Proddb). Execute the following command at the operating system level:

Windows: Create a shortcut to the following command:

```
full_pathname\sales.exe -dproddb
```

UNIX:

```
alias saleapp full_pathname/sales.exe –dproddb
```

VMS:

```
saleapp :== "$dir_spec sales.exe-dproddb"
```

Users can execute saleapp or sales (Microsoft Windows) to run the Sales Entry Application in the Proddb database, while the definition of the Sales Entry Application resides in the Testdb database.

As an alternate approach, copy the entire application from the Testdb database to the Proddb database using the copyapp utility, discussed in the section, Copying an Application into a Different Database. Then build a new executable image in the Proddb database.

Appendix E: Notes for Users of QUEL

This section contains the following topics:

QUEL Notes for the Building Applications Chapter (see page 497)
QUEL Notes for the ABF Development Example Chapter (see page 498)

For those who use QUEL as a query language with ABF and 4GL, this appendix covers QUEL syntax differences and gives alternate examples written with QUEL syntax.

Note: QUEL does not support global variables, record definitions, and constants. These variables are only supported by SQL.

For QUEL and 4GL reserved words, see the 4GL part of this guide.

QUEL Notes for the Building Applications Chapter

The sections below describe differences between QUEL and SQL as they apply to the topics discussed in Building Applications (see page 351).

Source Code File Names and Extensions for Procedures

The source code file names and extensions are listed below:

Operating System	QUEL File Extension	Non-QUEL File Extension	Programming Language
Windows	.osl	none	4GL/QUEL
	.qcb	.cob	EQUEL/COBOL
	.qc	.c	EQUEL/C
UNIX	.osl	none	4GL/QUEL
	.qc	.c	EQUEL/C
	.qcb	.cob	EQUEL/COBOL
	.qf	.f	EQUEL/FORTRAN
VMS	.osl	none	4GL/QUEL
	.qa	.ada	EQUEL/ADA
	.qc	.c	EQUEL/C

Operating System	QUEL File Extension	Non-QUEL File Extension	Programming Language
	.qcb	.cob	EQUEL/COBOL
	.qf	.for	EQUEL/FORTRAN

The Edit a Procedure Definition Frame

In the Edit a 4GL Procedure Definition frame, the Nullable field has a default value of "No" for a QUEL application.

Using EQUEL/Forms with ABF

You can perform most forms control procedures through 4GL. However, if you must use a form control that is not supported in 4GL, call an EQUEL procedure that displays a form. The table File Extensions in the previous section shows the languages for which EQUEL is provided.

If the EQUEL procedure uses a form which is not used by any user-specified frame, it must initialize the form with ###FORMINIT or ###ADDFORM. A compiled form that is not used by any user-specified frame must be specified in ING_ABFOPT1. For more information regarding this name, see Building Applications (see page 351).

The EQUEL procedure must use an EQUEL/Forms statement to display the form. The EQUEL code in EQUEL procedures cannot use ##INGRES, ##FORMS, ##EXIT, or ##ENDFORMS statements. For details, see the QUEL companion guide.

QUEL Notes for the ABF Development Example Chapter

The QUEL versions of two of the frames in the Sales Entry Application created in ABF Development Example (see page 453), appear in the following sections. The ABF development procedure is the same. Operations that use global variables, record definitions, and constants, are not included.

The 4GL Code for Topframe

The QUEL version of the 4GL code for the Topframe main menu follows:

```
"Sales" =
begin
    callframe salerep;
 end

"Customer" =
begin
    callframe customer;
 end


"New" =
begin
    callframe neworder;
 end

"End" =
begin
    callframe end;
 end
```

The 4GL Code for NewOrder

The QUEL version of the 4GL code for NewOrder is shown below. This code declares a menu with three operations: Add, Find, ListChoices and End. The first two operations contain examples of query statements (append and retrieve). ListChoices uses the built-in procedure help_field().

```
initialize (cnum = integer,
    rcount = smallint) =
begin
end

"Add" =
begin
    if custnum = 0 then
        message "You must enter a customer" +
            " number before selecting ADD";
        sleep 2;
        resume;
    endif;
    append to orders (product = product,
        custname = custname, custnum = custnum,
        quantity = quantity, currentdate = "today");
    append to marketing (prod = product,
        quantity = quantity);
    /* Check if a record exists in the table     */
    /* customer having the customer number in    */
    /* field "custnum" */
    neworder = retrieve (cnum = customer.number)
        where customer.number = custnum;
    /* See if a record already exists */
    inquire_ingres(rcount = rowcount)
    if rcount = 0 then
        append to customer (name = custname,
            number = custnum,
            address = "Whereabouts unknown");
    endif;
 end

"Find" =
begin
    if custnum = 0 then
        message "You must enter a customer" +
            " number before selecting FIND";
        sleep 2;
        resume;
    endif;
    /* Write the customer name into the    */
    /* "custname" field for the customer */
    /* number specified in the field "custnum"*/
    neworder = retrieve (customer.name)
        where customer.number = custnum;
    /* See if a row was found for that     */
    /* customer number */
    inquire_ingres(rcount = rowcount);
    if rcount = 0 then
         message "No such customer number";
        sleep 2;
        resume;
    endif;
    clear field product, quantity;
 end

/* ListChoices for field.  For the product     */
/* field this displays possible products,        */
/* taken from the Validation Check to        */
/* Perform field in the VIFRED Creating a        */
/* Form frame */
"ListChoices" =
begin
    callproc help_field();
        /*Built-in procedure */
end

"End", key frskey3 =
begin
    return;
 end
```

Appendix F: ABF Architecture

This section contains the following topics:

This appendix discusses the way ABF builds and stores applications. When you use the Create an Application operation, ABF sets up operating system structures and commands that determine:

- The directory structure used to store applications
- How applications are built

You must be familiar with object codes, libraries, linking, and directories before reading this appendix.

Where Applications Are Stored

ABF stores information about applications in the following locations:

- Source code directory
- Object code directory; this contains:
- Object files
  - Error listing files
  - Compiled form files
  - The extract file
- System catalogs

Each of these locations is discussed below.

Source Code Directory

When you create an ABF application, you specify the directory to hold the source code for frames, 4GL and host language procedures, and reports. ABF never deletes or purges files from this directory.

VMS: Vision purges obsolete versions of the generated 4GL code from the source code directory.

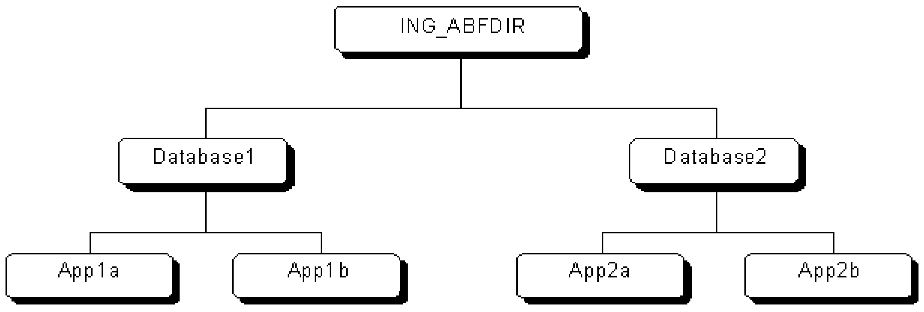
If you do not specify a source code directory, the default is the directory from which you started ABF. You can change the source code directory on the Application Defaults window; however, if you do this, you also must copy any source code files to the new source code directory.

Object Code Directory

ABF creates object files when it compiles the source code for application components such as frames, forms, and procedures. The logical/environment variable ING_ABFDIR points to the object code directory tree. Within this directory tree, ABF creates:

- A subdirectory for each database in which you have created applications
- A further subdirectory for each application that you have created within each database

The following figure illustrates the object code directory tree. Subdirectories within this tree always have the same name as the databases and applications they contain.



ABF automatically generates unique names for object files. These names are usually based on the name of the associated source code files and follow the pattern *objectname*.obj.

Windows: If the environment variable ING_ABFDIR points to a directory called abfhome, the pathname for the application's compiled object files is:

c:\abfhome\dbname\applicationname

UNIX: If the environment variable ING_ABFDIR points to a directory called abfhome, the pathname for the application's compiled object files is:

/**abfhome**/*dbname*/*applicationname*

VMS: If the logical ING_ABFDIR points to DISK1:[ABFHOME], the directory for the compiled files of the application is:

DISK1:[**ABFHOME**.*dbname.applicationname*]

ABF always purges files from the object code directory.

Error Listing Files

When a frame or procedure does not compile correctly, information about the errors is stored in the error listing file. This file has the name *componentname.lis*, where *componentname* represents the name of the frame or procedure.

The error listing file is stored in the object code directory. You can examine the file directly from the Error Listing frame.

Compiled Form Files

When ABF creates a compiled form while building an image, it creates file names as follows for the source code file:

*number***.c**

VMS:

*number***.mar**

where *number* is the object ID of the form, as stored in the ii_objects system catalog.

ABF also creates the file *number*.obj for the object code. In this case, both source and object files go into the object directory.

The Extract File

ABF uses the extract file to help build images. For an application that contains 3GL procedures, ABF also uses this file when you run the application with the Go operation. This file is called:

Windows:

abextrac.c

UNIX:

abextract.c

VMS:

abextract.mar

System Catalogs

ABF stores and maintains information about your applications in the Ingres system catalogs. These are tables in the same database as the application. See the *Database Administrator Guide* for a full description of these catalogs:

ii_objects

Contains a row with the following information about each application object (such as frames, forms, and reports) in the database:

- name
- owner
- object ID
- object class
- creation date
- modification date (this date is changed through a field on the Edit a User Frame Definition window).

ABF shares this catalog with Vision.

ii_encodings

Stores 4GL frames and procedures, forms, encoded in an Intermediate Language (IL). ABF uses the IL code when running an application with the Go operation. ABF shares this catalog with Vision and OpenROAD.

ii_locks

Manages concurrent user access to applications and application components. ABF shares this catalog with Vision and OpenROAD.

ii_longremarks

Contains the Long Remark text that you have specified for any application components. ABF shares this catalog with Vision and OpenROAD.

ii_abfclasses

Contains information about the attributes of ABF record types. ABF shares this catalog with Vision.

ii_abfobjects

Contains ABF-specific information (such as source code file and linker symbol) about ABF objects stored in the ii_objects catalog. ABF shares this catalog with Vision.

ii_abfdependencies

Describes the relationships between database objects (such as reports and forms) used in ABF applications. ABF shares this catalog with Vision.

ii_sequence_values

Used by the 4GL sequence_value function to generate surrogate keys for new rows in a table. ABF shares this catalog with Vision.

Forms in ABF Applications

This section discusses some of the ways in which ABF incorporates forms into applications.

How Forms and the Image Operation Work

If a User frame or Report frame uses a compiled form, then the following actions occur when you select the Image operation:

1. ABF compiles the form into a source file with the file extension .c (.mar on VMS) in the application's object code directory.
2. ABF compiles the source file into an object code file (with the .obj file extension).
3. ABF links the object code file into the executable when the application is imaged.
4. The form becomes part of the executable image and ABF no longer uses the form's definition stored in the system catalog.

If you specify that the frame does not use a compiled form, the form is fetched from the database each time the image is run. This is noticeably slower than using a compiled form. It also means that, if you want to run the image against another database, you must copy the image's forms to that database.

Linking Forms and 3GL Procedures

You must use the VIFRED Compile operation with forms that are called by a 3GL procedure. Then take the following steps to link the compiled form into the ABF application:

1. Create a linker option file (described later in this appendix).
2. Place the object files for these forms in the linker option file.
3. Either specify the linker option file in the ABF Application Default frame or define the ING_ABFOPT1 logical/ environment variable to point to the linker option file.
4. Create an image of the application.

The form's object files in the linker option file are linked into the executable image.

If the 3GL procedure uses a form that is also used by a user-specified frame, ABF already knows about the form. Do not list this form in the linker option file.

How an Application Is Built with the Go Operation

When you select Go to build an ABF application, ABF completes these steps:

1. Determines which components to include
2. Determines which frames to recompile
3. Compiles the application
4. Creates temporary files
5. Creates and compiles the extract file
6. Verifies the results of compilation
7. Runs the interpreter

The following sections discuss each of these actions.

Determining Which Components to Include

When you run an application with Go, ABF first determines which parts of the application to run:

- ABF runs the entire application if you select Go:
  - From the Applications Catalog
  - From the Frame Catalog, with the default start frame

- ABF runs only the frames and procedures in a specific frame tree if you select Go:
  - From the Frame Catalog, from the frame on which the cursor is placed
  - From the frame details window
  - In Vision, from the Visual Query window
- ABF includes all 3GL procedures in the application

ABF refers to the ii_abfdependencies catalog to determine which frames or procedures below the starting frame to call. In the following cases, frames or procedures do not have entries in the catalog:

- If you use a variable for the frame name in a callframe statement or for the procedure name in a callproc statement (except for 3GL procedures).
- If a called SQL procedure calls a 4GL frame or procedure using an exec 4GL statement.

In the above cases, you must compile these frames and procedures yourself.

Determining Which Frames to Recompile

In the second step, ABF determines which of the application components to recompile. It does this by checking:

- Flags

  ABF recompiles any frames or procedures for which a flag is set to mark the component as "out-of-date." This flag is set when you:
  - Change such data about the component as its source code file, form, or return type
  - Change the return type of a component called by this component
  - Change a global variable, global constant, or record that the component uses
  - For Vision frames, change the Visual Query or add escape code
- Dates

  If no flags are set, ABF recompiles components whose dates are more recent than the date of the Intermediate Language code stored in the ii_encodings catalog for the component. ABF checks the dates for:
  - Source code
  - Forms for frames
  - Tables used in type of table array and record declarations
  - Forms used in a type of form or type of table-field declaration

How Compiling the Application Works

In the third step, if recompilation is needed for a frame or procedure, then ABF:

- Generates IL code for the 4GL frames and procedures, and stores this code in the ii_encodings system catalog
- Compiles 3GL procedures into object files

How Temporary Files Are Created

In the fourth step, ABF creates a temporary file in which it creates a runtime table of frames and 4GL procedures to be run.

ABF deletes this file after the Go operation is complete.

How the Extract File Is Created and Compiled

In the fifth step, if the application that you are running contains any 3GL procedures, ABF creates an extract file in which it builds a runtime table of such procedures. This file is called:

Windows:

abextract.c

UNIX:

abextract.c

VMS:

abextract.mar

ABF then:

1. Compiles this file into an object file called abextract.obj.
2. Creates a temporary copy of the interpreter by linking this object file with the host language object files and libraries.

How Compilation Results Are Verified

In the sixth step, after ABF has performed the steps above, it proceeds based on the status of the compiled code:

- If there were no compilation failures, ABF runs the interpreter for the entire application or frame tree that you have specified (as described in the following section, Running the Interpreter).
- If ABF did not compile the starting frame or procedure or any 3GL procedure, then it cannot run the interpreter.
- If ABF did not compile any other frame or procedure, it gives you the option of running the interpreter for the rest of the application or frame tree.

If the user calls a frame or procedure that was not compiled, ABF displays the Unavailable Frame or Procedure window.

How Running the Interpreter Works

Finally, ABF runs the interpreter, passing to it the Intermediate Language code to run the application. If there are no 3GL procedures, the interpreter is the file:

Windows:

iiinterp, located in the **%II_SYSTEM%\ingres\bin** directory

UNIX:

iiinterp, located in the **$II_SYSTEM/ingres/bin** directory

VMS:

iiinterp.exe, located in the **II_SYSTEM:[INGRES.bin]** directory

If the application or frame tree includes any 3GL procedures, ABF uses the temporary copy of the interpreter that it created (as described in How the Extract File Is Created and Compiled (see page 508)). This file has the same name as the interpreter itself, and is stored in the application's object code directory. The file is deleted when you exit the application.

How an Application Is Built with the Image Operation

When you select Image to build an ABF application:

1. ABF includes all frames and procedures for the application image.
2. ABF determines which components to recompile.

   This is similar to the process described in the section, Building an Application with the Go Operation. Again, ABF checks the dates for the Intermediate Language (IL) code in the ii_encodings catalog.

   If a frame's IL code is not out-of-date, ABF also checks the object file. ABF recompiles the frame if the IL code is newer than the object code.
3. If recompilation is needed for a frame or procedure, ABF:
   - Generates IL code for 4GL frames and procedures
   - Generates C code from the IL code
   - Compiles this code and host language procedures into object files
4. For each compiled form in the application, ABF checks whether the form in the database is newer than the object file for the compiled form.

   If so, ABF creates a new compiled form file and then compiles it into an object file.
5. ABF generates the extract file and compiles it into the abextract.obj.
6. Finally, ABF links object files and libraries into an executable with the image name that you have specified.

Linking with Libraries

ABF uses a fixed set of libraries to link all applications. If your procedures require special libraries, add more libraries. ABF has a facility that allows you to designate a linker option file, which is included in the link.

You can do this in either of the following ways:

- Enter the name of the link options file in the Applications Default window
- Set the ING_ABFOPT1 logical/environment variable to the name of your link options file

If you set both values, the file name in the Applications Default window takes precedence.

Windows:

For example, if your procedure requires a library called userlib, you can create a file called myabf.opt as your own ABF linker option file. This file contains the full pathname for userlib:

```
c:\lib\userlib
```

Set the environment variable ING_ABFOPT1 to this pathname, as follows:

```
set ING_ABFOPT1=c:\lib\myabf.opt
```

When you image your application, ABF links in the library userlib.

Alternatively, you can specify c:\lib\myabf.opt as the link options filename in the Application Defaults window.

To add another library, for example c:\lib\newlib, you can modify myabf.opt. The file format for myabf.opt requires a single library or object module per line, as follows:

```
c:\lib\userlib
c:\lib\newlib
```

Because it is still set to myabf.opt, it is not necessary to change the link options file name in the Application Defaults window or redefine ING_ABFOPT1. Your application is now linked with both userlib and newlib.

UNIX:

For example, if your procedure requires a library called userlib, you can create a file called myabf.opt as your own ABF linker option file. This file contains the full pathname for userlib:

```
/usr/local/lib/userlib
```

Set the environment variable ING_ABFOPT1 to this pathname, as follows:

- C shell:

  ```
  % setenv ING_ABFOPT1 /usr/local/lib/myabf.opt
  ```

- Bourne shell:

  ```
  $ ING_ABFOPT1 = /usr/local/lib/myabf.opt
  $ export ING_ABFOPT1
  ```

When you image your application, ABF links in the library userlib.

Alternatively, you can specify "/usr/local/lib/myabf.opt" as the link options filename in the Application Defaults window.

To add another library, for example, "/usr/local/lib/newlib," you can modify myabf.opt. The file format for myabf.opt requires a single library or object module per line, as follows:

```
/usr/local/lib/userlib
/usr/local/lib/newlib
```

Because it is still set to myabf.opt, it is not necessary to change the link options file name in the Application Defaults window or redefine ING_ABFOPT1. Your application is now linked with both userlib and newlib.

VMS:

If your application uses a C procedure, you must link the application to the VAX C Runtime Library (VAXCRTL). Create a file that names this library and any other object modules to be included.

The file must be a VMS Linker option file and must have the extension .opt; for example, myabf.opt. The file myabf.opt contains the following line:

```
SYS$SHARE:VAXCRTL/SHARE
```

Define the logical name ING_ABFOPT1 as the full directory specification and filename of the file myabf.opt:

```
$ define ING_ABFOPT1 "dir_spec myabf.opt"
```

When you link your application, ABF links it with the VAX C Runtime Library. If you later decide that one of your procedures requires another library—called, for example, userlib—you can add it to myabf.opt. Place each single library or object module on a separate line. The file now contains the following lines:

```
SYS$SHARE:VAXCRTL/SHARE
dir_spec userlib/library
```

Include the string /library after the name of each library included in the option file.

Because it is still set to myabf.opt, it is not necessary to change the link options file name in the Application Defaults window or redefine ING_ABFOPT1. Your application is now linked with both the VAXCRTL and userlib.

For more discussion of linking with libraries, see Creating an Image of the Application (see page 429).

Updates to Linker Option Modules

When you choose the Go operation, ABF cannot check the dates on libraries included by the linker options file. When building an interpreter, ABF assumes that no modules in the linker file are updated during the current editing session. Thus, it does not relink with the new module.

If you update these modules, you must exit from ABF, or return to the ABF Edit an Application frame and reenter the application using the Edit or Go operation. This forces the updated modules to be linked.

The Image operation always relinks, so this procedure is not necessary when linker options modules change while you are building an application with Image.

Linking with Compiled Forms for Procedures

Use the linker option file to link any compiled forms that the host language procedures use in your application.

In the following example, the application uses the form "myform" compiled into the object file myform.obj, as well as the libraries mentioned in the above examples. (See *Character-based Querying and Reporting Tools User Guide* for details on VIFRED forms and more information on creating the object file.)

Edit the linker option file pointed to by ING_ABFOPT1 as follows:

Windows:

pathname\myform.obj
pathname\userlib
pathname\newlib

UNIX:

pathname/**myform.obj**
pathname/**userlib**
pathname/**newlib**

VMS:

dir_spec **myform.obj**
dir_spec **userlib/library**
dir_spec **vaxcrtl/share**

The application is now linked with the compiled version of the form "myform" as well as the two libraries.

Calling Library Procedures

To call a host language procedure for which you have the object code but no source code, complete the following steps to link an existing procedure into an application:

1. Add the name of the object file (or the library that contains the object code) to the linker options file.

   For example, to link a C procedure on VMS, use the VAX C Runtime Library.

2. Define a library procedure to ABF by creating a host language procedure.

   Use the same name for the procedure as you use in the callproc statement. The symbol field on the Edit a Procedure Definition frame must contain the same name as the procedure.

The list of object files and libraries specified in the default Linker Options field are included in the named procedure.

Appendix G: The ABF Demo Program

This section contains the following topics:

Your Ingres software includes a sample ABF application, the Project Management application. This appendix presents:

- Directions for installing and starting the application
- An overview of the contents of the application, including a map of the frames

The Project Management application is a complete database application you can examine and modify, and includes:

- A database populated with tables, forms, reports, and the sample application
- A subdirectory from your login directory containing all source code and any other files used by the application

Before you can run the Project Management application, you must install or reinstall it according to the procedures described in the following section. Make sure that Ingres itself is already installed.

Installing the Sample Application

Before you use the demonstration for the first time, follow these steps:

1. Read through and make sure you understand the preceding chapters and appendixes of this guide.
2. Talk to the system administrator to ensure that you are a valid user and have permission to create databases.
3. The application builds the subdirectory abfdemo in your login directory. If you already have a file or subdirectory called abfdemo in your login directory, rename it before installing the demonstration.

UNIX:

4. Set up the system commands for your use by adding the appropriate commands to your login set-up file:

   Bourne shell:

   Add the following lines to your .profile file:

   ```
   set PATH=$PATH:$II_SYSTEM/ingres/bin
   export PATH
   ```

   C shell:

   Add the following line to your .login file:

   ```
   set path=($path $II_SYSTEM/ingres/bin)
   ```

   You must log out and then log in again to activate the demo commands.

   Windows:

   No additional commands are needed.

   VMS:

   No additional commands are needed.

5. The demonstration application utilizes function keys. Set the environment variable term_ingres to the type of terminal you are using. See the appropriate sections of this guide for further information on this topic.

   For purposes of illustration, this guide illustrates Ingres sessions with term_ingres set to VT100.

Install ABFdemo

To install your copy of the ABF Demonstration

1. At the operating system prompt, type the following command:

   abfdemo

   You are prompted for a database name in which to store the Demonstration application.

2. Enter a database name. If a database already exists with the name you specify, you are prompted for another name.

   The name of the database is stored in the file iidb_name.txt in the abfdemo subdirectory containing your copy of the application (in your login directory) during the installation procedure.

   The installation program loads the database and copies files into the abfdemo subdirectory. The program asks you if you want to start using ABF immediately.

3. Type y and press Return.

   The ABF Applications Catalog appears. To use the application, you must compile it.

4. To compile the application, select Go from the menu.

   The initial compilation of the application takes several minutes. (Later, as you change modules of the application, only the modules you change are recompiled when you select Go.)

   After the compilation is complete, you can use or alter the application. You can modify both the application's sample database and the application itself exactly as if you had created them on your computer.

5. Select the Quit menu item from the ABF menu to return to the operating system.

Start the Sample Application

After you install the Demonstration Application, you can start ABF for your application at the operating system prompt.

To start the sample application

Enter the following command at the system prompt:

abfdemo

The abfdemo command recognizes that the application is installed and starts ABF using your application database.

To delete the sample application

1. *Before proceeding* with destruction of your database, copy any needed files from the *abfdemo* subdirectory.
2. At the operating system prompt, type the command:

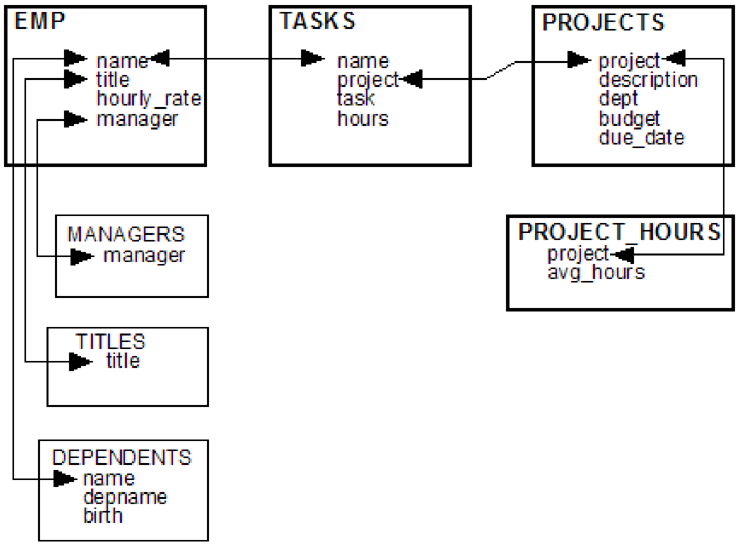
   deldemo

3. Press Return. ABF destroys your Demonstration database and application, including all the files in the *abfdemo* subdirectory, removing them from your login directory.

After using this command, you must reinstall the Demonstration application to run it again.

Contents of the Sample Application

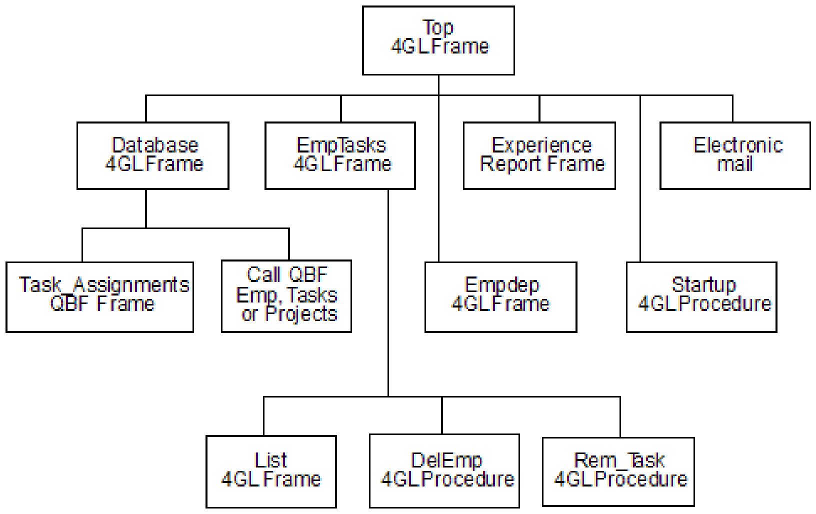
The ABF Demonstration application is a simplified program for tracking employees and their tasks. The database consists of the tables shown in the following figure.



The application consists of:

- 4GL frames
- Frames that run other Ingres tools (QBF and Report-Writer)
- 4GL, database, and 3GL procedures

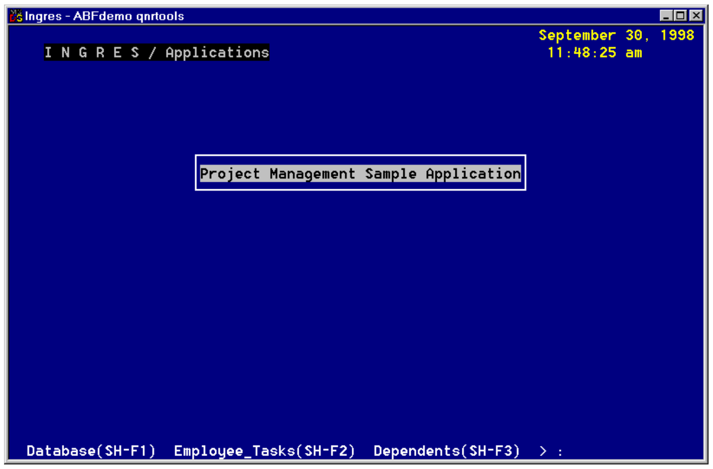
The basic structure of the Project Management application is shown in the following figure.



How to Run the Sample Application

To begin the demonstration, type abfdemo at the operating system prompt and press Return.

In a few moments, the application's introductory frame appears. The Introduction to Project Management Application frame, shown in the following figure, serves as an umbrella for the application and for some frames that let you see how the application works.


```
Ingres - ABFdemo qnrtools
                                                          September 30, 1998
    I N G R E S / Applications                             11:48:25 am

                      Project Management Sample Application

  Database(SH-F1)  Employee_Tasks(SH-F2)  Dependents(SH-F3)  > :
```

The introductory menu includes the following operations:

Database

Displays information about the database used in the Project Management application

Employee_Tasks

Uses 4GL to retrieve, manipulate and update data in the database

Dependents

Uses 4GL to demonstrate the use of hidden fields. The Dependents table field is displayed only when an employee has dependents.

Experience

Provides reports on the database

Mail

Provides access to electronic mail

The following function keys available for the application:

FRS key 1

Displays a help file about the introductory window

FRS key 2 or 3

Quits the application and return to the operating system

FRS key 16

Views the 4GL code for the top frame

The Top frame in the code listings that follow defines the operations of the introductory menu.

4GL Code for the Sample Application

This section contains the code used to create the frames and procedures of the Project Management application. If possible, read these listings as you review the individual frames in the window with ABF. In particular, it is valuable to examine each form with VIFRED as you read the 4GL code that accompanies it.

The map of the sample application presented earlier shows the relationship of these frames to one another. Here, they are presented in alphabetical order for convenience.

Windows: The pathnames for the application code are similar to the following examples:

```
'c:\usr\joe\abfdemo\database.hlp'
'c:\usr\joe\abfdemo\database.osq'
'c:\usr\joe\abfdemo\design.txt'
'c:\usr\joe\abfdemo\emptasks.hlp'
'c:\usr\joe\abfdemo\emptasks.osq'
'c:\usr\joe\abfdemo\list.osq'
'c:\usr\joe\abfdemo\top.hlp'
'c:\usr\joe\abfdemo\top.osq'
```

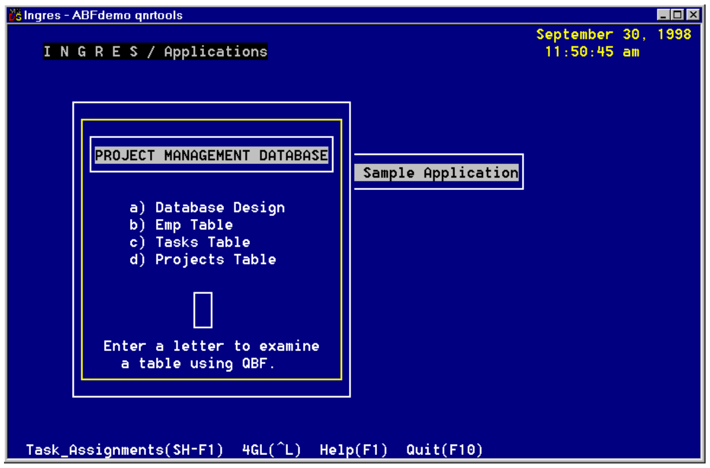
UNIX: The pathnames for the application code are similar to the following examples:

```
'usr/joe/abfdemo/database.hlp'
'usr/joe/abfdemo/database.osq'
'usr/joe/abfdemo/design.txt'
'usr/joe/abfdemo/emptasks.hlp'
'usr/joe/abfdemo/emptasks.osq'
'usr/joe/abfdemo/list.osq'
'usr/joe/abfdemo/top.hlp'
'usr/joe/abfdemo/top.osq'
```

VMS: The example file names are as shown in the application code.

Database Frame

The Database Frame shown in the following figure is a user-specified frame coded in 4GL. With this frame, the user can use QBF to examine the Emp, Tasks, and Projects tables from the database.



```
Ingres - ABFdemo qnrtools
                                                      September 30, 1998
   I N G R E S / Applications                          11:50:45 am

        PROJECT MANAGEMENT DATABASE
                                        Sample Application
          a) Database Design
          b) Emp Table
          c) Tasks Table
          d) Projects Table

        Enter a letter to examine
          a table using QBF.

 Task_Assignments(SH-F1)  4GL(^L)  Help(F1)  Quit(F10)
```

The single field in the form uses the field activation capabilities of 4GL to select a table with a single keystroke. The Task_Assignments menu operation starts QBF with a Join Definition between the Emp and Tasks tables. The 4GL source code for this frame follows.

```
/* database.osq */
field 'selection' =
begin
set_forms field '' (normal(selection)=1);
 if selection = 'a' then
helpfile 'Database Design'
 'USER:[JOE.ABFDEMO]design.txt';
 elseif selection = 'b' then
```

```
call qbf (qbfname = 'emp', mode = 'retrieve');
 elseif selection = 'c' then
call qbf (qbfname = 'tasks', mode = 'retrieve');
 elseif selection = 'd' then
call qbf (qbfname = 'projects', mode = 'retrieve');
 else

set_forms 'field' " (blink(selection)=1);
 message 'Selection must be a, b, c, or d'
with style=popup;
 endif;
 end
 'Task_Assignments' =
begin
callframe task_assignments;
 end
 '4GL', key frskey16 =

begin
helpfile 'Database Frame 4GL'
 'USER:[JOE.ABFDEMO]database.osq';
 end
 'Help', key frskey1 =
begin
help_forms(subject='Database Information',
 file = 'USER:[JOE.ABFDEMO]database.hlp);
 end
 'Quit', key frskey3 =
begin

callproc timer_on (secs = 2);
 return;
 end
key frskey2 =
begin
exit;
 end
on timeout =
begin
callproc timer_off;
 end
```

DelEmp Procedure

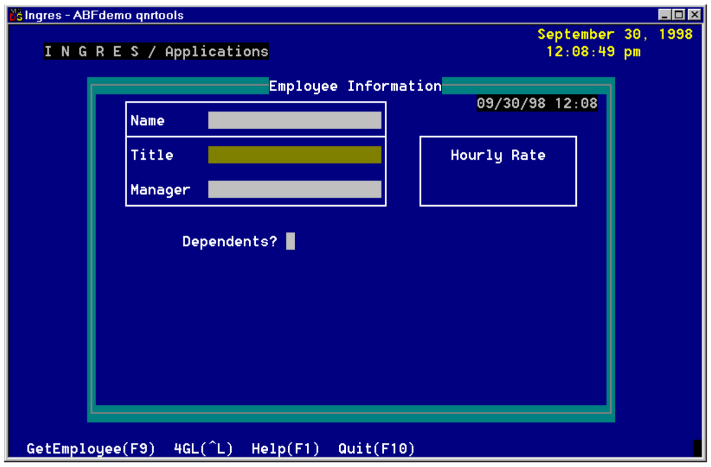
The DelEmp 4GL procedure isolates the deletion of records from the Emp and Tasks tables within the application to control the logical relationships between data in separate tables. Data is deleted within a transaction to ensure that the database is left in a logically consistent state.

```
/* Delete an Employee and all that */
/* employee's tasks from the database */
procedure del_emp (empname=varchar(20)) =
begin
message 'Deleting employee ' + empname +
        '.  .  .';
 delete from tasks where name = :empname;
 delete from emp where name = :empname;
 delete from dependents where name = :empname
commit;
 end
```

EmpDep Frame

The EmpDep Frame, shown in the following figure, lets you query the Employee table to display data about specific employees. The frame demonstrates the use of hidden fields and invisible fields.

When a user selects the GetEmployee menu operation, the frame checks the Dependents table. If the employee has dependents, then the "dependents" table field is displayed and the hidden field "deps" is set to y; otherwise the table field is invisible and "deps" is set to n.



The following is the 4GL source code for this frame:

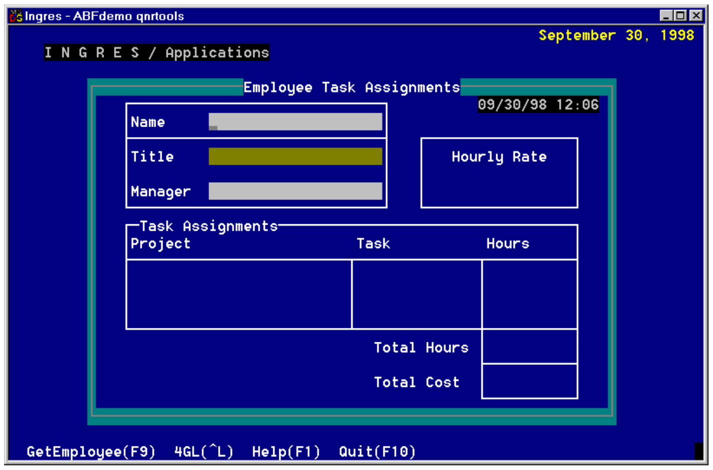
```
/* * Empdep.osq *
** View and Update Emp and Tasks information.
 **
*/

initialize (resp = c3, curr_field = c32)=
 begin
mode query;
 today = date('now');
 callproc timer_on(secs=15);
 end
GetEmployee, key frskey4 =



Begin
empdep = select name, title, manager, hourly_rate
from emp
where qualification(name=name,title=title,
 manager=manager, hourly_rate=hourly_rate)
 dependents = select distinct *
from dependents
where name=:name
order by birth
begin
initialize =


begin
if :dependents[1].depname = " then
set_forms field "
 (invisible(dependents)=1);
 deps = 'N';
 else
set_forms field "
 (invisible(dependents)=0);
 deps = 'Y';
 endif;
 redisplay;
 end
 'Next', key frskey4 =
begin

message 'Retrieving next Employee.';
 next;
 if :dependents[1].depname = " then
set_forms field "
 (invisible(dependents)=1);
 deps = 'N';
 else
set_forms field "
 (invisible(dependents)=0);
 deps = 'Y';
 endif;
 redisplay;
 end
 'Quit', key frskey3 =
begin
```

```
endloop;
 end
on timeout =
begin
today = date('now');
 end
end;
 set_forms field ''
 (invisible(dependents)=1);
 clear field all;
 today = date('now');
 redisplay;
 end
 '4GL', key frskey16 =
begin

helpfile 'Employee-Dependents Frame 4GL'
 'USER:[JOE.ABFDEMO]EMPDEP.OSQ';
 end
 'Help', key frskey1 =
begin
help_forms (subject = 'Employee Information',
 file = "USER:[JOE.ABFDEMO]EMPDEP.HLP');
 end
 'Quit', key frskey3 =
begin
callproc timer_on(secs=2);
 return;
 end


key frskey2 =
begin
exit;
 end
on timeout =
begin
today = date('now');
 end
```

EmpTasks Frame

The EmpTasks Frame shown in the following figure demonstrates how you can use 4GL to retrieve, manipulate,
and update data in the database.

The initial menu operation, GetEmployee, queries the database based on values entered into fields by the user and displays a submenu of operations to manipulate the retrieved data.


Ingres - ABFdemo qnrtools
September 30, 1998
I N G R E S / Applications
Employee Task Assignments
09/30/98 12:06
Name
Title
Manager
Hourly Rate
Task Assignments
Project
Task
Hours
Total Hours
Total Cost
GetEmployee(F9) 4GL(^L) Help(F1) Quit(F10)

This is the 4GL source code file for the frame:

```
/*
 * Emptasks.osq
 *
 * View and Update Employee information and
 * Tasks assignments.
 *
 */

initialize (resp = c3, curr_field = c32, fnkey=i4)=
 begin
        mode query;
        today = date('now');
        callproc timer_on(secs=15);
 end

GetEmployee, key frskey4 =
Begin
  emptasks = select name, title, manager, hourly_rate
        from emp
        where qualification(name=name,title=title,
            manager=manager, hourly_rate=hourly_rate)
    tasktable = select distinct *
         from tasks
        where name=:name
        order by task
```

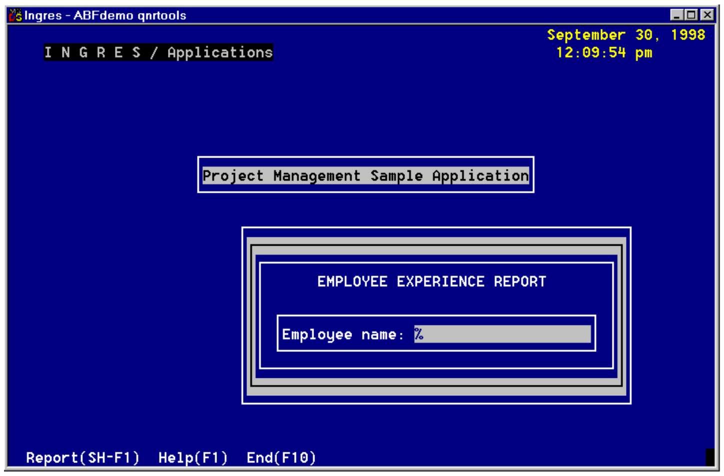
```
begin
    initialize =
    begin
        callproc sum_hours(hourly_rate);
        redisplay;
    end
    field hourly_rate, field tasktable.hours =
    begin
        callproc sum_hours(hourly_rate);
        redisplay;
        resume next;
    end
    'Next', key frskey4 =

    begin
        message 'Retrieving next Employee.';
        next;
        callproc sum_hours(hourly_rate);
        redisplay;
    end
    DeleteEmp =
    Begin
     resp = prompt 'Type "y" to delete '+name+'.'
            with style=popup;
        if lowercase(left(resp,1)) = 'y' then
         callproc del_emp(empname = name);
         next;
        else
         message 'Employee not deleted.';
         sleep 2;
        endif;
        redisplay;
    end

    RemoveTask =
    Begin
        inquire_forms field emptasks
             (curr_field = name);
        if curr_field = 'tasktable' then
            callproc rem_task(person = name,
                proj = tasktable.project,
                job = tasktable.task);
            deleterow tasktable;
            callproc sum_hours(hourly_rate);
        else
            message 'Place cursor on task'
                        + ' to be deleted'
            with style=popup;

        endif;
    end
```

```
              UpdateEmp =
              Begin
                  message 'Updating ' + name + ' .   .   .   ';
                      update emp e set
                          name = :name,
                          title = :title,
                          manager = :manager,
                          hourly_rate = :hourly_rate
                          where :name =
                          any (select name from tasks t
                              where t.name = e.name);
                          unloadtable tasktable
                      begin
                          update tasks set hours =
                          :tasktable.hours
                      where :tasktable.task = task and
                          :name = name and
                          :tasktable.project = project;
                      end;
                  commit;
              end

              'Help', key frskey1 =
              begin
                  help_forms (subject = 'EmpTasks Frame',
                  file = 'USER:[JOE.ABFDEMO]EMPTASKS.HLP');
              end
              '4GL', key frskey16 =
              begin
                  callproc timer_off();
                  helpfile 'EmpTasks Frame'
                      'USER:[JOE.ABFDEMO]EMPTASKS.OSQ';
                  callproc timer_on(secs=15);
              end

              'Quit', key frskey3 =
              begin
                  endloop;
              end
              on timeout =
              begin
                  today = date('now');
              end
          end;
          clear field all;
          set_forms field " (blink(tot_hours)=0); /* turn off
              blinking */
          today = date('now');
          redisplay;
       end
```

```
field 'title', field 'manager' =
begin
    if lowercase(title) = 'help' or lowercase(manager) =
        'help' then
        inquire_forms field ''
            (curr_field=name);
        if curr_field = 'title' then
            title = callframe list
                (list.info = 'Titles';
            list.vals = select distinct val = title
                    from titles order by val);
    else
            manager = callframe list
                (list.info = 'Managers';
            list.vals = select distinct val = manager
                    from managers order by val);
        endif;
    endif;
    resume next;
 end

key frskey2 =
begin
    inquire_forms field ''
            (curr_field=name);
    if curr_field = 'title' then
        title = callframe list
            (list.info = 'Titles';
        list.vals = select distinct val = title
                from titles order by val);
    else
        manager = callframe list
            (list.info = 'Managers';
        list.vals = select distinct val = manager
                from managers order by val);
    endif;
    resume next;
 end

 '4GL', key frskey16 =
begin
    callproc timer_off();
    helpfile 'EmpTasks Frame 4GL'
     'USER:[JOE.ABFDEMO]EMPTASKS.OSQ';
    callproc timer_on(secs=15);
 end
 'Help', key frskey1 =
begin
  help_forms(subject = 'Employee Task Assignments Frame',
  file = 'USER:[JOE.ABFDEMO]EMPTASKS.HLP');
 end
 'Quit', key frskey3 =
begin
    callproc timer_on(secs=2);
    return;
 end
on timeout =
begin
    today = date('now');
 end
```

Experience Report Frame

The Experience frame is a Report frame that displays a pop-up form in which the user enters parameters. The Experience Report uses the Report-Writer to construct a cross-reference query. The text of the report file is contained in the following figure:



```
   I N G R E S / Applications                              September 30, 1998
                                                            12:09:54 pm

                  Project Management Sample Application

                         EMPLOYEE EXPERIENCE REPORT

                         Employee name: %

 Report(SH-F1)  Help(F1)  End(F10)
```

```
/* Example of exchanging rows and */
/* columns in a report */
.Name    Experience
.ShortRemark
    Employee Experience Crossreference Report
.LongRemark
    7/1/88 Created
.EndRemark

.Query     select distinct name, task
    from tasks
    where name like '$name' or name = '$name'
.Sort name
.Head report
    .Newpage
```

```
.Head page
    .Newline
    .Underline .Center .Print
        "Employee Assignments Summary Report"
    .Nounderline
    .Newline2
    .Center .Print date ("today")
        (d "February 3, 1901")
    .Newline4
    .Center 50 .Print "Assigned Tasks"
    .Newline2
    .Underline .Print "Name"
    .Tab 25 .Print "Design"
    .Tab 35 .Print "Implement"
    .Tab 48 .Print "Test"
    .Tab 58 .Print "Debug"
    .Tab 68 .Print "Manage"
    .Nounderline
    .Newline2

.Head name
    .Newline
    .Print name(c23)
.Detail
    .If task = "Design" .Then
        .Tab 25 .Print "Yes"
    .Elseif task = "Implement" .Then
        .Tab 35 .Print "Yes"
    .Elseif task = "Test" .Then
        .tab 48 .Print "Yes"
    .Elseif task = "Debug" .Then
        .Tab 58 .Print "Yes"
    .Elseif task = "Manage" .Then
        .Tab 68 .Print "Yes"
.Endif

.Foot page
    .Need 3
    .Newline 1
    .Center .Print page_number("zn")
    .Newline 1
```

List Frame

The List frame is a pop-up form that allows the user to enter a value from a list generated by a query to the database. The table field in the form is filled by passing an SQL query as a parameter in the call to this frame.

```
/* list.osq */
/* list all managers or all job titles */
on Select_value, key frskey4 =
begin
    if vals.val IS NULL then
        message 'No value selected.';
    else
        message vals.val + ' selected.';
    endif;
    return vals.val;
 end


 'No_selection', key frskey3 =
begin
    message 'No value selected.';
    return ";
end
 '4GL', key frskey16 =
begin
    helpfile 'List Frame 4GL'
        'USER:[JOE.ABFDEMO]list.osq');
 end
on timeout =
begin
    callproc timer_off;
 end
```

RemTask Procedure

The Remtask 4GL procedure isolates the deletion of records from the tasks table from the application. Additional qualifications coan be enforced by this procedure, because it controls access within the application.

```
/* remtask.osq /

/ remove a task from the project management
** database */

procedure rem_task (person=c20, proj=c12,
    job=c9) =
begin
    message 'Removing ' + :proj + ' Task: '
        + job;
    sleep 3;
    delete from tasks where
        tasks.name = :person and
        tasks.project = :proj and
        tasks.task = :job;
 end
```

Startup Procedure

The Startup procedure initializes one of the FRS keys to be used with the 4GL menu operation in all the frames in the application. Simply changing the definition in this procedure can change the actual key used by the application. Throughout the application, Frskey 16 refers to this key.

```
procedure startup =
begin
        set_forms frs (map(frskey16)= 'controlL');
        set_forms frs (label(frskey16) = '^L');
 end
```

SumHours Procedure

The SumHours 3GL procedure calculates the time and cost for projects. The following is the C code for this procedure:

```
/**
  **  Sum_hours.sc:  Sum  the  total  hours  in
  **  the  tasktable.hours  column  in  emptasks.
  **
  **  Arguments:
  **  hourly_rate:  the  current  hourly  rate;
  **
  **  Side  Effects:
  **  Computes  and  displays  total  hours
  **   and  cost  in  emptasks  form.
  **  Changes  tot_hours  to  blink  if  over
  **   40  hours  assigned.
  **
  **  Returns:
  **  Nothing.
  **/
void

sum_hours(hourly_rate)
 double      hourly_rate;
 {
exec  sql   begin  declare  section;
    long    hours,  tot_hours;
    double  tot_cost;
    short   ni,  overhours,  state;
 exec  sql   end  declare  section;
    tot_hours  =  0;  overhours=  0;
 exec frs  unloadtable emptasks tasktable(:hours:ni=hours,
  :state=_STATE);
 exec  frs         begin;
          if  (state  !=  4  &&  state  !=0
            &&  ni  !=  -1)
          tot_hours  =  tot_hours  +  hours;
 exec  frs         end;
    tot_cost  =  tot_hours  *  hourly_rate;
 exec  frs         putform  emptasks
 (tot_hours=:tot_hours,tot_cost=:tot_cost);
    if  (tot_hours  40)
      overhours=1;
 exec  frsset_frs  field  emptasks  (blink(tot_hours)  =
:overhours);
    return;
 }
```

Timer_On and Timer_Off Procedures

The application uses the timer_on and timer_off 4GL procedures at various points to control the screen display when the user provides no input, as follows:

- timer_on sets the timeout interval at 2 seconds
- timer_off redisplays the time in the upper right corner of the frame, if the user has not performed any action for 2 seconds

The following is the code for these 4GL procedures:

```
procedure timer_on (secs = i2)
 begin
        set_forms frs (timeout = secs);
 end
procedure timer_off =
begin
        set_forms frs (timeout = 0);
 end
```

Top Frame

The Top frame of the application is a simple menu frame that allows users to access various parts of the application. It appears in the 3rd figure of this appendix. The fields Today and Current_time display the current date and time. The startup procedure can be used to specify any FRS or other commands that must be executed when the application begins.

The following is the 4GL code for this frame:

```
/*
 *
top.osq
 *
A simple main application frame.
 *
 *
 */


initialize =
begin
    message 'Starting the ABF Demonstration Application.  .  .';
        today = date('today');
        callproc startup;
        callproc timer_on(secs=2);
 end

on timeout =
begin
        current_time=date('now');
 end
Database =
Begin
        callframe database;
 end

Employee_Tasks =
Begin
        set_forms field '' (invisible(current_time)=1);
        redisplay;
        callframe emptasks;
        set_forms field ''
            (invisible(current_time)=0);
 end

Dependents =
Begin
        callproc timer_off;
        callframe empdep;
        callproc timer_on(secs=2);
 end
```

```
Experience =
Begin
        callproc timer_off;
        callframe experience;
        callproc timer_on(secs=2);
 end

Mail =
Begin
        call system 'mail';
 end
'4GL', key frskey16 =
begin
        callproc timer_off;
        helpfile 'Top Frame 4GL'
            'USER:[JOE.ABFDEMO]top.osq';
        callproc timer_on(secs=2);
 end

 'Help', key frskey1 =
begin
        callproc timer_off;
        help_forms (subject = 'Top Frame Help Information',
            file = 'USER:[JOE.ABFDEMO]top.hlp');
        callproc timer_on(secs=2);
 end
 'Quit', key frskey2, key frskey3 =
begin
        exit;
 end
```

Appendix H: Roles for ABF Applications

This section contains the following topics:

This appendix discusses the way that roles are built into ABF and Vision applications.

Roles are used to associate database permissions with an application. You can use roles to give the application its own permissions, beyond those of the user who runs it. This adds flexibility in assigning and controlling privileges. For example, an application can be assigned a role, such as "employeechanger," with privileges that allow changes to be made to the database. Because the user does not normally have these privileges, this ensures that the user can only make changes to the database by running the application. You can build your application with any necessary checks to ensure that changes to the database are made in an appropriate way.

To associate a password with the application role, ABF prompts you to give the password when you test the application or create an executable image. This password is not saved anywhere in the database, although ABF remembers the password from the previous session. The password is never displayed on your screen.

After you create an executable image with a role, the password is not necessary to run the application image.

Roles for Applications

ABF and Vision allow you to assign a default role for an application. The default role must be an existing role, created through SQL.

The default role is assigned through the ABF Application Defaults frame and stored in the application's catalog records. You can override the default role in the following ways:

- Use the -R flag to assign a role to an application when building an image with the imageapp command. If you use imageapp to build an application that runs under a role, you are prompted for the role's password. After you create an executable image with an application role, the password is not necessary to run the application image.
- Enter a different role in the ABF Build Application Image pop-up form when building an image from within ABF or Vision.

The application's default role is not used in these two cases:

- If you use copyapp to copy an application with a default role into an installation that does not have the Knowledge Management Extension, the default role is copied into the catalogs, and is not used.
- If you use Ingres Net for an installation that does not have the Knowledge Management Extension to use an application that has a default role, the default role is not used.

Note: The Knowledge Management Extension is part of Ingres.

See the *Database Administrator Guide* for further information about roles.

PART 4: Embedded Forms Programming

Chapter 15: Embedded Forms Program Structure

This section contains the following topics:

This chapter describes the general structure of an embedded forms-based program. Forms statements enable you to create a forms-based application. Forms can be created using Vision, ABF, or the VIFRED. (For instructions on creating forms using VIFRED, see *Character-based Querying and Reporting Tools User Guide.*)

The sequence of operations in a typical forms-based application is as follows:

1. Invoke the FRS. FRS controls the display whenever a form is displayed as part of an application.
2. Declare one or more forms to the FRS.
3. Display the first form, initiating a display loop that the FRS maintains while the form is displayed.
4. Offer the user a choice of different operations and specify code to be executed when these operations are chosen. When the user chooses an operation, the display loop is suspended, the specified code is executed, and then the display loop is resumed.
5. Provide the user with the option to exit the program, for example, by selecting an End or Quit operation, which terminates the display loop. When the display loop is terminated, the program can display another form or disconnect from the FRS.

You can use host language variables to specify portions of many of the forms statements. For a general discussion of how to use host language variables in an embedded SQL or QUEL program, see your query language reference guide; for information about how an individual statement uses host language variables, see the specific statement descriptions in Forms Statements (see page 613). The examples in this guide use embedded SQL. For QUEL examples, see EQUEL/FORMS Examples (see page 757).

General Syntax of Forms Statements

Forms statements require the following syntax.

ESQL/FORMS:

```
[margin] exec frs forms_statement [terminator]
```

EQUEL/FORMS:

```
[margin] ## forms_statement [terminator]
```

If you are programming in embedded SQL, the key word **frs** replaces the key word **sql** following **exec**. If you are programming in embedded QUEL, you must precede forms statements with **##** (as you do for EQUEL statements).

For information about the *margin* and the *terminator*, see your query language reference guide and host language companion guides.

Note: The statement syntaxes in this guide omit the **exec frs** or **##** portions of the syntax; you must supply them according to the query language you are using. The examples use SQL conventions, except in EQUEL/FORMS Examples (see page 757), where QUEL conventions are used.

Basic Structure of a Forms-based Program

The following example presents a simple forms-based program. This example illustrates the fundamental forms statements and their relative positions in the program.

```
exec sql include sqlca;
 exec sql begin declare section;
 namevar character_string(20);
 salvar  float;
```

```
exec sql end declare section;
 exec sql whenever sqlerror stop;
 exec sql connect personnel;
 exec frs forms;
 exec frs forminit empform;
 exec frs display empform;
exec frs initialize (ename = :namevar, sal = :salvar);
 exec frs activate menuitem 'Help';
exec frs begin;
 program code;
exec frs end;

 exec frs activate menuitem 'Add';
exec frs begin;
 program code;
exec frs end;
 exec frs activate menuitem 'End';

exec frs begin;
 exec frs enddisplay;
exec frs end;
 exec frs finalize;
 exec frs endforms;
 exec sql disconnect;
```

The section of code beginning with the statement display and ending with finalize constitutes the *display block* for the form empform. The statement in the display block specifies the operations available with the form. In the preceding example, the activate menuitem statements create a menu line that looks like this:

```
Help Add End
```

You can also display forms that have no associated operations. Under certain circumstances, a program can display a form without even being connected to a database. It is not a requirement of the FRS that a forms application needs a database connection. Generally, however, forms applications include database operations and so require connection to a database.

How the FRS Is Invoked

Before you can use a form in your embedded query language program, you must start the FRS by issuing the forms statement. The FRS depends on the user having defined the type of terminal being used so that screen and cursor management can be performed correctly. (See the *Character-based Querying and Reporting Tools User Guide* for information about terminal definition.) After issuing the forms statement, your application can declare the forms to be used in the application.

How Forms Are Declared

You create forms using the VIFRED utility. After creating a form, you have the option of compiling the form (in VIFRED). Uncompiled forms can be changed without recompiling the application that uses them, but the application must retrieve the form definition when the form is declared. Compiled forms are linked into the application, eliminating the need to retrieve the form definition at run time, but the application must be recompiled if the form is changed.

To use a form in an application, you must declare the form to the FRS using the forminit (for uncompiled forms) or addform (for compiled forms) statement, as explained below.

forminit Statement—Declare Uncompiled Forms

To declare an uncompiled form, use the forminit statement.

This statement has the following syntax:

```
forminit :formname;
```

The forminit statement retrieves the form definition from the forms catalogs in the database; therefore, it must be issued after the program connects to the database. The forminit statement is an executable statement, and must be placed outside any declaration sections.

addform Statement—Declare Compiled Forms

To declare a compiled form, use the addform statement.

This statement has the following syntax:

```
addform :formname;
```

To create a compiled form, you must define and compile it in VIFRED, compile the resulting form into object code, and finally, link the object code with the application program. Whenever you change a compiled form's definition in VIFRED, you must recompile the form and relink the application in order for the changes to appear in your program.

In the application program, you must declare a host variable for the address of the form definition; the following is an SQL example:

```
exec sql begin declare section;
  external integer    formname;
 exec sql end declare section;
```

Formname must be the name given the form in VIFRED.

You must specify a colon in front of *formname*, because the addform statement is referencing the variable that contains the form's address, not the form name itself. When the *formname* is used in later statements, it is not necessary to precede it with a colon. See your host language companion guide for information about the exact format and data type acceptable for your host language.

display Statement—Display a Form

After the FRS has been invoked and at least one form declared, the program can display the form by issuing the display statement.

This statement has the following syntax:

```
display formname [mode]
    with style=fullscreen[(screenwidth =
 current|default|narrow|wide)]
      popup[(option=value {, option=value})]];
```

The display statement directs the FRS to display the specified form on the screen in the specified mode and style. The form remains on the screen until the user takes some action to end its display or until another form is displayed on top of it. While the form is displayed, a display loop is in effect. The FRS manages the display and generally maintains control of the application. The user can examine and place data in fields and activate various operations. In response to actions taken by the user, control is transferred from the FRS to sections of program code defined as operations. After the operations are complete, control returns to the FRS, where it remains until another operation is activated or the loop is ended.

Display Modes

Forms can be displayed in the following modes:

fill

Displays data on the screen and allows the data to be modified by the user. (Fill mode is the default if no mode is specified in the display statement.) Data displayed when the form first appears can come from one of two sources:

- Default values specified in the form's VIFRED definition
- Values placed in fields by the initialize statement

When you display a form in fill mode, all fields on the form are cleared before any initialization code is executed. Any values left in the form's fields from a previous display are cleared.

update

Displays whatever data is presently in the form. Unlike fill mode, fields in the form are not cleared before display. The user can change the data.

read

Specifies that the user can examine but not change the data on the form

query

Allows the user to enter a comparison operator (for example, = or <), in addition to the data. Using these operators, your application can construct queries. The form must allow extra spaces in the fields for operators.

Display Styles

To display a form in either full-screen or pop-up style, specify the style clause. The style clause overrides the form's style definition specified in VIFRED.

If your form is specified in VIFRED as full screen with a specified screenwidth setting and you display the form as a pop-up, the FRS ignores the screenwidth setting.

The Fullscreen Option

To display a form over the entire screen, specify style = fullscreen. To change the effective screen size of your terminal for the form's display, specify the optional screenwidth clause. Whenever the terminal display width is changed, the FRS clears the screen before changing the width. Changing the screenwidth also changes the size of the menu line.

If you issue a display statement that changes the terminal display width, the change occurs when the form is actually displayed, after the initialize block for the form, if any, is executed. If the display is terminated from the initialize block, no change takes place.

For an explanation of each of the screenwidth options, see the display statement description in Forms Statements (see page 613).

The Popup Option

To display a form on top of the currently displayed form, specify style = popup. The *options* parameters specify the position of the pop-up and its border style. To specify the position of a pop-up dynamically at run time, you can use host variables to specify *options*. (For a complete discussion of the pop-up options, see the display statement description in Forms Statements (see page 613).)

If you omit the *options*, runtime defaults override the values specified in VIFRED. By default, the pop-up is displayed with a single-line border, and a location as close as possible to the current field (the field in which the cursor is located when the pop-up is displayed.)

If you display a form as a pop-up whose form definition is fullscreen with a specified screenwidth, the style specified in the display statement overrides the screenwidth option.

Other Display Attributes

To set field display attributes such as reverse video, blinking, underlining, and intensity, use the set_frs statement. You can also use set_frs to change a specific field to display only or to make it invisible (see Invisible Fields (see page 607) for more information).

You cannot set some display attributes for derived fields. (A derived field is a field whose value depends on values in another field or fields in the form. For more information, see Derived Fields (see page 608).)

For complete information about the set_frs statement, see set_frs Statement—Set FRS Features (see page 724).

The Display Block

The display statement is followed by groups of statements that define the operations associated with the displayed form. The statements associated with the display of a single form constitutes that form's display block. The statements in the display block are divided into sections, each of which performs a separate function. There are three types of sections:

The initialize section

This section initializes the form. It begins with the initialize statement, which must be located immediately after the display statement. The initialize section can transfer data into the form when the form is first displayed. It can also include a block of initialization code that is executed when the form is first displayed.

The activate section

This section provides an operation, which the user can choose to run. A display block can contain numerous activate sections. Each activate section begins with the activate statement and includes a block of code that is executed when the user performs the specified action. This action can include selecting a menu item, pressing a specified key, or simply failing to perform any action for a specified period of time (an activate timeout).

The finalize section

This section must be the last in the display block. It contains only the finalize statement. The finalize statement can be used to transfer data from the form into program variables immediately prior to the end of the display loop. Unlike the other sections, no block of code can be associated with it.

All the sections are optional but, if included, must appear in the order described. A display block can include many activate sections, but only one initialize and one finalize section. The initialize and activate sections can have associated *blocks of code*, bounded by the exec frs begin and exec frs end statements (exec frs is specific to SQL). You can place any appropriate host language or embedded query language statement between the begin and end statements.

All statements within a display block must be part of an initialize, activate, or finalize section. A statement appearing outside a section indicates the end of the display block. For this reason, no embedded query language statements, host language statements, or comment lines can appear between the sections.

A display loop begins when the display block is encountered in the program's execution. At that time, the initialize section is executed, the form is displayed, and control is transferred from the program to the FRS. Control remains with the FRS until the user interrupts the FRS by selecting an operation, for example, a menu item. At that point, control transfers to the operation's activate section and the statements in that section are executed. Once the activate section has been executed, control returns to the FRS. When the user selects an operation that contains a statement to end the display, the finalize section is executed, the display loop ends, and control transfers from the FRS to the first statement in the program that follows the end of the display block.

Display loops can be nested; you can include a display block within an activate section of another display block. Thus your application can allow a user to call another form by selecting a menu item, for example. When the inner display loop ends, control passes back to the outer display loop and the original form reappears on the screen. A form can call a form that calls another form, and so on. The only restriction is that an already displayed form cannot be displayed again; that is, you cannot nest a display of the same form.

The Initialize Section

The initialize statement appears in the display block immediately after the display statement. Its primary purpose is to transfer data from program variables into simple fields on the form. It is executed once, when the display loop is started.

This statement has the following syntax:

```
initialize [(fieldname = data {, fieldname = data})];
 begin;
     initialization code;
 end;
```

The *fieldname* is the name of the simple field into which *data* is put. For example, the statement:

```
exec frs initialize
   (namefield = :namevar, salfield = :salvar);
```

takes values from two program variables and places them in simple fields when the form is first displayed. The *data* can also be a constant value. For example, the statement:

```
exec frs initialize
   (namefield = 'bill', salfield = 20000.00);
```

The field name can also be specified as a string variable, which allows the program to determine in what field to place the data:

```
exec frs initialize
   (:fieldvar = 'bill', salfield = 20000.00);
```

In all statements that use a field name, the field name must be the field's *internal* name, as specified in VIFRED, and does not necessarily bear any relationship to the field's title, which usually appears with the field on the form.

By associating a block of code with the initialize statement, you can specify an initialization operation before the form is displayed.

Activation Operations

Although you can create a simple forms application with only the initialize and finalize statements that allows the user to perform simple data entry and editing, most applications require more sophisticated control over the form display. For instance, it is possible to provide the user with help in filling out a form before the form is complete, let the user retrieve and update data from the database during the form's display, and catch values that the user has entered in two fields of the form to determine the value of a third field.

For such needs, you can use activate blocks to create operations. When the user selects an operation during the form's display, the statements in the operation's activate block are executed. The form remains on the screen after the section has been executed, allowing the user to continue entering and editing data or to select another operation.

In its simplest form, the activate section has the following syntax:

```
initialize [(fieldname = data {, fieldname = data})];
 begin;
     initialization code;
 end;
```

When the specified *condition* occurs, the statements in the section are executed. You can specify more than one *condition* in a single activate section. However, if you specify more than one activation for the same condition, only the last one in your source file is executed.

Seven types of conditions, corresponding to the seven types of user-specified operations, are available. Two of these are used exclusively with table fields and are discussed in Table Field Activation Operations (see page 595). The others are:

menuitem

Causes the accompanying block of code to be executed when the user selects a menu item. The activate statement for this condition has the following syntax:

```
activate menuitem menuname;
```

The *menunames* appear on the menu line at the bottom of the screen, ordered according to the sequence of activate menuitem statements in the display block.

before field

Specifies that the statements associated with this condition are executed whenever the user moves the screen cursor into a specified field or, if you specify all, any field on the form. (This is often called an entry activation.) An activate statement for this condition has the syntax:

```
activate before field fieldname|all;
```

[after] field

Specifies that the statements associated with this condition are executed whenever the user moves the screen cursor out of a specified field or, if you specify all, any field on the form. (This is often called an exit activation.) An activate statement for this condition has the syntax:

```
activate [after] field fieldname|all;
```

frskey

Specifies that activation for this condition occurs when the user presses the control or function key mapped to the designated FRS key. (The *Character-based Querying and Reporting Tools User Guide* explains FRS (FRS) key mapping in detail.) You can specify the FRS key using a literal or an integer variable. The FRS key number must be in the range of 1 to 40. The syntax for this condition is:

```
activate frskeyN |frskey :integer_variable |frskey : integer_constant;
```

timeout

Specifies that the program code associated with this block is executed whenever a display loop times out. A display loop times out when a user does not perform any keyboard activity within a specified time limit. An activity can include such actions as moving the cursor, entering data, or making a menu choice. The syntax for this condition is:

```
activate timeout;
```

The following example illustrates the various types of activate statements described above:

```
...
 exec sql begin declare section;
    namevar   character_string(20);
    salvar    float;
exec sql end declare section;
 exec sql connect personnel;
exec frs forms;
 exec frs forminit empform;
exec frs display empform;
exec frs initialize (ename = :namevar, sal = :salvar);

 exec frs activate before field 'ename';
exec frs begin;
    program code;
exec frs end;
 exec frs activate frskey5;
exec frs begin;
    program code;
exec frs end;

 exec frs activate menuitem 'Help';
exec frs begin;
    program code;
exec frs end;
 exec frs activate menuitem 'Add';
exec frs begin;
    program code;
exec frs end;

 exec frs activate menuitem 'End', frskey3;
exec frs begin;
    exec frs enddisplay;
exec frs end;
 exec frs activate field 'salary';
exec frs begin;
    program code;
exec frs end;

 exec frs activate timeout;
exec frs begin;
    program code;
exec frs end;
 exec frs finalize;
 exec frs endforms;
exec sql disconnect;
...
```

The first activate section in the example is executed when the user moves the cursor into the field ename. The user can execute the operation in the second section by pressing the function or control key that is mapped to FRS key 5. The next three activate sections are associated with menu items. They cause the following menu line to appear at the bottom of the form:

```
Help Add End
```

The user can execute the last menu item activate section in the example by either selecting the menu item End or pressing the function key mapped to FRS key 3, illustrating the fact that a single activate section can specify more than one condition. After the menu item activate section is a field activation that is executed when the user moves out of the salary field. If the user fails to perform any keyboard action within some previously specified time limit, the timeout activate section is executed.

Submenus

Submenus, which are menus that are not directly linked to a displayed form, can be used as a more detailed menu inside a display loop or even as a multi-item prompt without an attached form. The Ingres Forms subsystems often use submenus to provide greater detail for users. For example, while editing a field with VIFRED, your menu can change without changing the form you are editing. Submenu sections are signaled by the submenu statement. That statement is then followed by activate menuitem or activate frskey sections. You cannot include field activations in a submenu because submenu does not allow you to move the cursor off the menu line while the submenu is displayed.

The example below illustrates a submenu that displays when the user chooses the Help menu item:

```
...
exec frs activate menuitem 'Help';
exec frs begin;
     exec frs submenu;
     exec frs activate menuitem 'Application';
     exec frs begin;
          provide help about current application;
     exec frs end;
     exec frs activate menuitem 'Form';

     exec frs begin;
          provide help about current form;
     exec frs end;
     exec frs activate menuitem 'MenuOperations';
     exec frs begin;
          provide help about current menu operations;
     exec frs end;
exec frs end;
...
```

When the user selects the Help menu operation, the current menu line is replaced by this submenu:

```
Application Form MenuOperations
```

When the user selects one of the submenu operations, the statements in its activate section are executed. The original menu line then returns.

A submenu can also include an activate timeout section.

Nested Menus

A nested menu is invoked by the display submenu statement.

This statement has the following syntax:

display *submenu*;

Nested menus differ from simple submenus in the following ways:

- You can define field activations (in addition to menu items and FRS keys) in nested menus. Field activations in the outer display loop (or nested menu) are not inherited.
- Running from a nested menu is exactly like running from a display loop; this includes using the resume statement. Unlike submenus, nested menus allow the user to move the cursor around the screen the same way as in a display loop.
- Multiple levels of nesting are allowed.
- To terminate a nested menu, you must use either the breakdisplay or enddisplay statement. The behavior of these statements are the same as in a normal display loop. Breakdisplay simply terminates the nested menu while enddisplay first validates all fields before exiting. Once a nested menu exits, the enclosing display loop (or nested menu) is restored.

The following embedded SQL example shows how you can use display submenu to create a nested menu:

```
...
exec frs activate menuitem 'EmpOps';
exec frs begin;
     exec frs display submenu;
     exec frs activate menuitem 'UpdateAge';
     exec frs begin;
          Do processing to update employee's age;
          exec frs breakdisplay;
     exec frs end;
     exec frs activate menuitem 'UpdateSal';
     exec frs begin;
          Do processing to update employee's salary;
          exec frs breakdisplay;
     exec frs end;
     exec frs activate menuitem 'End';
     exec frs begin;
          exec frs breakdisplay;
     exec frs end;
exec frs end;
```

How Returning from an Operation Works

An activate section can end when all of its associated statements have been executed, or it can end when a resume, enddisplay, or breakdisplay is executed.

In some instances, the operation can end if a timeout occurs. The timeout feature lets the programmer specify how long the FRS can wait for keyboard input from the user. If the user fails to supply some keyboard input within the specified time, then the display loop times out and exits. Control returns to the statement unless the display block contains an activate timeout block.

Going Back to the FRS

When an activate section ends, control returns to the FRS, unless an enddisplay or breakdisplay is issued. When control returns to the FRS, the FRS positions the cursor on the same field that it was on prior to the operation. If there are any entry activations defined for the field, this default behavior does not cause them to be performed.

If the default behavior is not satisfactory, you can change the behavior with the resume statement. You can place the cursor exactly where you want it when an operation ends or you can cause an entry activation operation when the cursor returns to the original field.

For example, it is not appropriate to allow an operation activated by the after field condition to use the default because each time the user tried to move on to another field, the operation executes and the cursor returns to the same field. To solve this problem, use the resume statement as the last executable statement in the operation:

```
resume next;
```

Use resume next only in an activate field or activate column section.

The various resume statements are as follows:

resume entry

Positions the cursor on the same field that it was on before the operation and causes any entry activations defined for that field to be performed

resume next

Continues the prior action. This is often used to ensure that an operation, such as a cursor movement, which initiates a field activation, is completed when the field activation is finished.

resume field *fieldname*

Positions the cursor on the specified field

resume column *tablename columnname*

Positions the cursor on the specified column of the specified table field

resume menu

Positions the cursor on the menu line. You cannot use this resume statement to position the cursor on a simple, non-nested submenu (that is, a menu displayed with the submenu statement).

resume nextfield

Positions the cursor on the first accessible field that is next in the tab sequence

resume previousfield

Positions the cursor on the first accessible field that is before the current field in the tab sequence

How You Terminate the Display Loop

You can terminate a display loop by issuing an enddisplay or breakdisplay from within an operation or you can create a separate operation that ends the display. For example, the following activate block creates an End operation that contains an enddisplay to end the display loop:

```
exec frs activate menuitem 'End', frskey3;
 exec frs begin;
     exec frs enddisplay;
exec frs end;
```

The enddisplay terminates the display loop after the FRS performs a validation check on the data in the form. (Validation criteria can be specified for fields when the form is defined in VIFRED. See How Field Validation Works (see page 609) for more information about validations.) If there are no errors in the data, the FRS executes the finalize statement (if any) and then transfers control to the first statement following the end of the display block in the program. If invalid data is found, the display loop continues, with the cursor resuming on the field in error, if possible. (It is not possible if the invalid data was found in an invisible or display-only field.)

To terminate the display loop without validating the data in the form or executing the finalize statement, use the breakdisplay statement. This statement is useful if, for example, you want to discard the data entered on the form.

Finally, if a display block contains no activate statements, the runtime user terminates the display loop when the Menu key is pressed by the user. However, if the display block contains any activate menuitem or frskey statements, the program must provide an explicit means for the user to exit from the display loop.

How the Timeout Feature Works

The timeout feature enables you to control how long the forms system waits for requested input from a user in a forms-based application. If the user fails to perform some keyboard action or respond to a prompt or message in the specified time, the forms system terminates the FRS statement that is waiting for input.

To set the timeout period use the set_frs statement. Once set, the timeout period applies to all display loops, prompts, messages, and all error messages requiring a carriage return for confirmation. The period remains in effect until the forms session is finished or another set_frs statement is issued to change the period's length. The specifying statement can be inside or outside of a display loop, or inside a nested display loop; once executed, it affects the application globally. (If inside a display block, the statement must be inside an activate section.)

Timeouts set in an application do not, however, affect any procedures called by the application. If you want to impose timeout limits on called procedures, you must do so in the code of the called procedure itself. Additionally, timeouts set in an application have no effect on any Ingres tools called by your application using the call statement.

By default, when a display loop times out, the loop exits as if a breakdisplay had been issued. No field validations are performed and the finalize statement is not executed. As an alternative, you can place an activate timeout block in the loop, which executes whenever the display loop times out. Use this block to ensure that the display loop is exited properly, and that all open transactions, display loops, and select loops are correctly terminated. (See the activate statement description for more information about activate timeout blocks.)

If the timeout occurs outside of a display loop while the forms system is waiting for a response to a prompt or message, the system behaves as if the user had typed a carriage return. You can use the inquire_frs statement with the command constant to determine if the user's exit is due to an actual carriage return or a timeout.

putform and getform Statements—Transfer Data During a Display Loop

A forms program must be able to put data into and get data from a field within a display loop. The initialize and finalize statements put and get data, respectively, only as the form display begins and ends. To put or get data at any other time use the putform and getform statements.

These statements have the following syntax:

```
putform [formname] (fieldname = data {, fieldname = data});

getform [formname] (variable = fieldname
      {, variable = fieldname});
```

As you can see, these statements correspond closely to the initialize and finalize statements. See Forms Statements (see page 613), for complete information about these statements.

finalize Statement—Transfer Data at the End of a Display Loop

The display block usually ends with a finalize statement. The primary purpose of this statement is to transfer data from the form's fields into program variables at the end of the display loop.

This statement has the following syntax:

```
finalize [(variable = fieldname {, variable = fieldname})];
```

Values from the *fieldnames* are transferred into the variables. The finalize statement executes when the display loop terminates unless the termination resulted from a breakdisplay statement or a timeout.

endforms Statement—Disconnect from the FRS

To sever the connection to the FRS, issue the endforms statement at the end of your application.

This statement has the following syntax:

```
endforms;
```

You can connect to and disconnect from the FRS only once in an application.

An Extended Example

The following example presents an embedded application, designed to illustrate some common features of forms-based applications. This example uses the employee table in the personnel database. The employee table has the following description:

```
exec sql declare employee table
     (eno    integer2 not null,
     ename   char(20) not null,
     age     integer1 not null,
     sal     float4   not null,
     dept    integer2 not null);
```

The form used in the application, empform, contains fields that correspond in name and type to the columns in the employee table.

The application allows the runtime user to retrieve an employee's data based on a unique employee number and to update or delete the data.

```
exec sql include sqlca;
 exec sql begin declare section;
     eno        integer;
     ename      character_string(20);
     age        integer;
     sal        float;
     dept       integer;
     queried    integer;
     /* Flag indicates "Query" menu item selected */
     found      integer;
     /* Flag indicates some rows were found */
exec sql end declare section;
```

```
 /*
** STOP on database errors,
** but CONTINUE on database warnings and
** NOT FOUND conditions.
 */
exec sql whenever sqlerror stop;
 exec sql connect personnel;
 exec frs forms;
 exec frs forminit empform;
exec frs display empform;

 exec frs initialize;
exec frs begin;
     queried = 0; /* No "Query" selected yet */
exec frs end;
 exec frs activate menuitem 'Help';
exec frs begin;
     exec frs submenu;
     exec frs activate menuitem 'Application';
     exec frs begin;
          /* provide help about current application */
          exec frs help_frs (subject = 'Application',
               file = 'applicat.hlp');
     exec frs end;

     exec frs activate menuitem 'MenuOperations';
     exec frs begin;
          /* provide help about current menu operations */
          exec frs help_frs (subject = 'Menu',
               file = 'menu.hlp');
     exec frs end;
exec frs end;

 exec frs activate menuitem 'Query';
exec frs begin;
     exec frs getform empform (:eno = eno);
     if (eno = 0) then
          exec frs message
               'You must enter an employee number 0';
          exec frs sleep 2;
          exec frs resume field eno;
     end if;
     found = 0;

     exec sql select ename, age, sal, dept, 1
          into :ename, :age, :sal, :dept, :found
          from employee
          where eno = :eno;
     if (found = 0) then
          exec frs message 'No employee found';
          exec frs sleep 2;
          exec frs clear field eno;
     else
          exec frs putform empform
               (eno = :eno, ename = :ename, age = :age,
               sal = :sal, dept = :dept);
          queried = 1;
     end if;
exec frs end;
 exec frs activate menuitem 'Update';
exec frs begin;
     if (queried = 0) then
          exec frs message
               'You must Query before you Update';
          exec frs sleep 2;
          exec frs resume;
     end if;

     exec frs validate;
               /*Confirm data is valid before updating */
     exec frs getform empform
          (:eno = eno, :ename = ename, :age = age,
          :sal = sal, :dept = dept);
     exec sql update employee
          set ename = :ename, age = :age, sal = :sal,
                       dept = dept
          where eno = :eno;
     exec sql commit;
     queried = 0;
exec frs end;

 exec frs activate menuitem 'Delete';
exec frs begin;
     if (queried = 0) then
          exec frs message
          'You must Query before you Delete';
          exec frs sleep 2;
     else
          exec frs getform empform (:eno = eno);
          exec sql delete from employee where eno = :eno;
          exec sql commit;
          queried = 0;
     end if;

exec frs end;
 exec frs activate menuitem 'End', frskey3;
exec frs begin;
     exec frs breakdisplay;
exec frs end;
 exec frs finalize;
exec frs endforms;
 exec sql disconnect;
```

For more extensive examples of forms applications, consult your host language companion guide.

Dynamic FRS

Dynamic FRS enables you to write forms-based applications that can be used with any form. Like Dynamic SQL, Dynamic FRS uses the SQL Descriptor Area (SQLDA) as a storage space for the information returned at run time by the describe statement. For information about Dynamic SQL, see the *SQL Reference Guide*. The Dynamic FRS describes statement returns descriptive information about either the fields on a form or the columns in a table field.

Note: Dynamic FRS is not available in QUEL. For dynamic behavior in QUEL use the param statement. See the *QUEL Reference Guide.*

Dynamic FRS also provides the using clause extension to give dynamic capabilities to several FRS input and output statements such as getform and loadtable. For a list of statements, see the section about the using clause. Unlike Dynamic SQL, Dynamic FRS does not allow you to prepare a statement for execution or description.

The following is a fragment of a Dynamic FRS program. This fragment takes data from a form specified by the user at runtime and inserts it into a database table whose name and description is the same as the form:

```
exec sql include sqlda;
 exec frs prompt ('Enter form name:' :name_var);
exec frs forminit :name_var;
 exec frs describe form :name_var into sqlda;
 analyze the sqlda, inspecting sqltype and sqllen;
allocate variables and set sqldata and sqlind;
 build Dynamic SQL insert statement using sqlname
fields and place it in the insert_buffer;


exec sql prepare ins from :insert_buffer;
 exec frs display :name_var;
exec frs activate menuitem 'Insert';
exec frs begin;
     exec frs getform :name_var using descriptor sqlda;
     exec sql execute ins using descriptor sqlda;
exec frs end;
...
```

The Describe Statement and Dynamic FRS

The describe statement returns data type, length, and name information about the fields of a form or the columns of a table field to the SQLDA. The application uses this information by means of the using clause extensions of some forms input and output statements. (For a full description of the use of the describe statement in Dynamic FRS, see The Using Clause in Dynamic FRS (see page 568).)

The describe statement can be issued at any time after the form is initialized. It is valid to issue the describe statement from inside a display loop, even if the form described is the form being displayed.

exec frs describe form Statement—Describe a Form

This statement describes a form.

This statement has the following syntax:

```
exec frs describe form formname [mode]
 into descriptor_name;
```

formname

Specifies the name of an initialized form

mode

Determines what subset of fields is described. *Mode* must be one of the following:

update

Specifies that the FRS returns the description of all updatable fields in a form. Display-only or query-only field descriptions are not returned. This mode corresponds to a form displayed in update or fill mode.

query

Specifies that the FRS returns the description of updatable and query-only fields. This mode corresponds to a form displayed in query mode.

all

Specifies that the FRS returns a description of all fields in the form, including those that are display-only or query-only. You can use this mode to map complete database table descriptions to a form. This is the default mode.

Regardless of the *mode* selected, the FRS always returns the data type and name of any table field encountered in the form. For information about how to handle table fields, see Processing the SQLDA in Dynamic FRS (see page 566).

After describing a form, the descriptor (SQLDA) contains the data type, length, and name descriptions of all the form fields corresponding to the specified *mode*.

exec frs describe table Statement—Describe a Table Field

This statement describes a table field.

This statement has the following syntax:

```
exec frs describe table formname tablename [mode]
  into descriptor_name;
```

formname

Specifies the name of an initialized form

tablename

Specifies a table field within the associated form (*formname*). You do not need to initialize a table field before describing it.

mode

Specifies which subset of columns is described. The mode values and the column subsets they specify are identical to those of the describe form statement. For example, if the query mode is specified when describing a table field, the descriptor contains the description of all updatable and query-only columns.

descriptor_name

Contains the data type, length, and name of all the columns in the table field corresponding to the specified mode. The FRS does not return hidden column names nor the internal row variables, _state and _record. If you want to use these variables in an application, the application must explicitly create a sqlvar element for each.

How SQLDA Is Processed in Dynamic FRS

Like Dynamic SQL, Dynamic FRS uses the SQLDA to store the results of a describe statement. However, in Dynamic FRS, the interpretation of the content of the sqld and sqlvar elements differs slightly:

- The **s**qld indicates the number of form fields or table field columns associated with the describe statement (rather than the number of result columns).

  If the value of **s**qld is zero, then the form had no fields or the table field had no columns of the given mode. If sqld is greater than sqln, then the program must reallocate the SQLDA to provide more storage buffers and reissue the describe statement. When the SQLDA is used to dynamically set or retrieve information from a form or table field, sqld must be equal to the number of form fields or table field columns affected by the statement. When used in a FRS application, the SQLDA components contain the following information.

- Each sqlvar element describes one of the form's fields or one column of the described table field. Each sqlvar element consists of the following:

  sqltype

  Specifies a 2-byte integer indicating the type of the field or column

  sqllen

  Specifies a 2-byte integer indicating the length of the field or column

  sqldata

  Specifies a pointer to the host variable described by the type and length

  sqlind

  Specifies a pointer to the indicator variable associated with the host variable

  sqlname

  Specifies the field or column name

  All other elements of the SQLDA are identical, whether used in Dynamic FRS or Dynamic SQL statement.

With two exceptions, the procedures for analyzing and processing the information in the SQLDA are the same in Dynamic FRS as in Dynamic SQL. These exceptions follow.

A Dynamic FRS Describe Form Statement Returns a Type Code of 52

This code indicates that the described field is a table field. You must issue a describe table statement to collect the column information about the table field. You can issue this statement either at the end of the processing or immediately. If you issue the describe table immediately, you must use a different SQLDA.

For example, the following program fragment, which generates a report from a form, checks the field's data type and, if a table field type is encountered, allocates another SQLDA and describes the table field into it.

```
...
 exec frs describe form :form_var into :form_desc;
 /* Confirm that form descriptor is large enough */
if (form_desc.sqld > form_desc.sqln) then
     free form_desc;
     allocate a descriptor with sqld or more
                          sqlvar elements;
     exec frs describe form :form_var
                    into :form_desc;
end if;

 /* Generate report for form */
call report_form(form_var, form_desc);
 /* Find all table fields (type 52),
** describe them and generate a report */
for index = 1 to form_desc.sqld loop
     if (form_desc.sqlvar(index).sqltype = 52) then
         table_var = form_desc.sqlvar(index).sqlname;
         exec frs describe table :form_var :table_var
                        into :table_desc;

         /* Confirm that table field descriptor
         ** is large enough */
         if (table_desc.sqld > table_desc.sqln) then
             free table_desc;
             allocate a descriptor with sqld or more
                                 sqlvar elements;
             exec frs describe table :form_var :table_var
                                 into :table_desc;
          end if;
          call report_table(table_var, table_desc);
     end if;
end loop;
...
```

Dynamic FRS Uses the sqlname Field Differently than Dynamic SQL

In Dynamic FRS, you must retain the value in **s**qlname to execute subsequent statements using the descriptor. The FRS uses these values to associate the fields or columns with the variables pointed at by sqldata. Dynamic SQL associates database columns with the sqldata variables based on the order of the columns in the SQL statement.

The Using Clause in Dynamic FRS

The using clause gives the FRS input and output statements their dynamic capabilities. The using clause enables these statements to use the descriptive information in the SQLDA to insert into and retrieve data from the dynamically named form or table field.

This clause has the following syntax:

```
using [descriptor] descriptor_name;
```

Dynamic FRS substitutes this clause for the parameter list in the following statements:

getform
putform
getrow
putrow
insertrow
loadtable
unloadtable

For example, the dynamic versions of the getform, insertrow, and unloadtable statements are:

```
exec frs getform [formname]
 using [descriptor] descriptor_name;

exec frs insertrow formname tablename [row]
 using [descriptor] descriptor_name;

exec frs unloadtable formname tablename
 using [descriptor] descriptor_name;
exec frs begin;
 program code
exec frs end;
```

Unlike the using clause associated with the Dynamic SQL execute statement, which can use the SQLDA or host variables, the Dynamic FRS using clause can only use the SQLDA.

The rules regarding the statement objects, *formname*, *tablename*, and *row*, do not change when the using clause is used.

When an input statement (putform, putrow, insertrow, and loadtable) uses the SQLDA, the SQLDA must contain field or column names with corresponding data areas having the values to be assigned into the form or table. For output statements (getform, getrow, and unloadtable), the SQLDA must contain field or column names with corresponding data areas to receive the information retrieved from the form or table.

The names entered into sqlname must be valid field or column names. When you use the SQLDA to retrieve from or assign into a table field, sqlname can also refer to table field hidden columns or internal state variables. However, these column and variable names are not returned by the FRS describe statement. You must allocate a sqlvar element for each individually.

To retrieve a query operator entered into the specified field or column, set sqlname to getoper(*fieldname*) or getoper (*colname*). The associated result variable must be an integer.

Examples of the Using Clause

The following examples show the use of the using clause and the getoper function. Read your host language companion guide for information about how to modify and or place values in the SQLDA.

This example demonstrates the use of the SQLDA to insert data into a form. It is the dynamic equivalent of the statement:

```
exec frs putform :form_var (age = :i4_var,
  comment = :c100_var:indicator_var);
```

Example:

```
exec sql include sqlda;
 declare the constants "int=30" and "char=20"
/* Assume that form has no more than 10 fields */
allocate an SQLDA with 10 elements;
 sqlda.sqln = 10; /* Number of allocated vars */
sqlda.sqld = 2; /* Number of vars used */

/* 4-byte integer to put into field "age" */
sqlda.sqlvar(1).sqltype = int;
sqlda.sqlvar(1).sqllen = 4;
sqlda.sqlvar(1).sqldata = address(i4_var);
sqlda.sqlvar(1).sqlind = null;
sqlda.sqlvar(1).sqlname = 'age';
 /* 100-byte nullable character
** to put into field "comment"
*/

sqlda.sqlvar(2).sqltype = -char;
sqlda.sqlvar(2).sqllen = 100;
sqlda.sqlvar(2).sqldata = address(c100_var);
sqlda.sqlvar(2).sqlind = address(indicator_var);
sqlda.sqlvar(2).sqlname = 'comment';
 i4_var = 1234;
c100_var = 'This is a comment.';
null_indicator = 0;               /* Not null */
exec frs putform :form_var using descriptor sqlda;
```

This second example demonstrates a simple routine that displays a form, gets values, performs calculations on the values and puts them back into the form. The form is dynamically specified. This example assumes that there are no table fields on the form.

```
exec sql include sqlda;
 exec frs prompt ('Enter form name:', :form_var);
exec frs forminit :form_var;
 allocate an sqlda;
 exec frs describe form :form_var into sqlda;
 /* Allocate variables, retaining field names */
allocate_result_variables(sqlda);
 exec frs display :form_var;
 exec frs activate menuitem 'Calc';


exec frs begin;
     exec frs validate;
     exec frs getform :form_var using descriptor sqlda;
     perform_calculations(sqlda);
     exec frs putform :form_var using descriptor sqlda;
exec frs end;
 exec frs activate menuitem 'End';
exec frs begin;
     exec frs breakdisplay;
exec frs end;
 exec frs finalize;
```

This third example demonstrates the modification of the sqlname field to include the getoper function to retrieve the query operator of a field displayed in query mode. The fragment shown here assumes that the SQLDA has been included and allocated in the program. This example corresponds to the following hard-coded statement:

```
exec frs getform :form_var (:op_var = getoper(sal),
   :f8_var:indicator_var = sal);
...
```

Example:

```
/* 4-byte integer query operator of field "sal" */
sqlda.sqlvar(1).sqltype = int;
sqlda.sqlvar(1).sqllen = 4;
sqlda.sqlvar(1).sqldata = address(op_var);
sqlda.sqlvar(1).sqlind = null;
sqlda.sqlvar(1).sqlname = 'getoper(sal)';

 /* 8-byte nullable floating-point from "sal" */
sqlda.sqlvar(2).sqltype = -float;
sqlda.sqlvar(2).sqllen = 8;
sqlda.sqlvar(2).sqldata = address(f8_var);
sqlda.sqlvar(2).sqlind = address(indicator_var);
sqlda.sqlvar(2).sqlname = 'sal';
 exec frs getform :form_var using descriptor sqlda;
```

Application-level Mapping

See the *Character-based Querying and Reporting Tools User Guide* for more detailed information on Key Mapping.

An application created with 4GL or one of the embedded query languages can use its own set of mappings. These application-level mappings take precedence over all other mappings. Application-level mappings can be specified either *statically* or *dynamically*.

Static Mapping

To use *static* application mapping an application mapping file can be read in at any point during the running of an embedded query language application with the following statement:

Windows:

```
set_frs frs (mapfile = full_pathname\file_name)
```

The parameter *full_pathname\file_name* is the full pathname and file name for the mapping file.

4GL uses a similar statement to read in a mapping file:

```
set_forms frs (mapfile = full_pathname\file_name)
```

UNIX:

```
set_frs frs (mapfile = full_pathname/file_name)
```

The parameter *full_pathname***/***file_name* is the full pathname and file name for the mapping file.

4GL uses a similar statement to read in a mapping file:

```
set_forms frs (mapfile = full_pathname/file_name)
```

VMS:

```
set_frs frs (mapfile = file_specification)
```

The *file_specification* parameter is the full specification for the mapping file.

4GL uses a similar statement to read in a mapping file:

```
set_forms frs (mapfile = file_specification)
```

When a mapping file is read in while the applications runs, it is merged with the set of previous mappings. New mappings overwrite existing mappings where they overlap. An application-level mapping file can be specified more than once during the running of an application.

Dynamic Mapping

An application can also perform *dynamic* key mappings through another set_frs (set_forms for 4GL) statement using the key word map. This statement can create or change mappings for one or more function, control, or arrow keys at any point during the running of the application. For example, the following 4GL statement equates PF3 with FRS key 5, so that any key activation operation using FRS key 5 is activated when the end user presses PF3:

```
set_forms frs (map(frskey5) = 'pf3');
```

The new mappings take precedence over conflicting mappings at any level.

How You Implement Keys in an Application

The process of creating forms applications using an embedded query language consists mainly of defining operations that a user can invoke by pressing a key to manipulate data on a form. Certain operations, such as control of cursor movement, are fundamental to the use of forms and have been predefined in the FRS; these are the FRS statements. Because they are predefined, you can give a user access to any of them by mapping the FRS command to a function or control key (or arrow key in a UNIX or VMS environment) within a mapping file. The user can then press the mapped key to invoke the operation.

For situations in which the predefined FRS statements cannot accomplish the task, you must define an operation within the application and map it to a function key, control key, or arrow key. Two steps are involved:

1. Define an operation in the code and specify the method(s) by which the user can activate it. Only certain types of operations can be invoked by a key.
2. Create a mapping file that is called by the application, which maps the operation to a function key or control key (or to an arrow key in UNIX or VMS environments).

There are three activation methods:

Field activation

Activates an operation by leaving a specified field

Menu item activation

Activates an operation by selecting a menu item on the menu line. If you also map the menu item (by position) to a function, control, or arrow key in the mapping file, the end user can also invoke the operation by pressing the key mapped to that menu item's position, any time that particular key mapping is in effect.

FRS key activation

Activates an operation by pressing a function key, control key, or arrow key mapped to a FRS key. A FRS key operation does *not* appear as an item on the menu line, unless the application makes the FRS key operation synonymous with a menu item operation.

Any combination of these activation methods can be specified for a single operation. For example, you can define an operation that the end user can activate either by leaving a particular field or by pressing a function key. The statement that defines forms operations in the embedded query language is the activate statement (see your embedded query language reference guide). Following are examples of the two types of operations that, in combination with a mapping file, permit activation by control or function keys of menu item operations and FRS key operations. The examples have each been written in SQL and 4GL. See the section that applies to your operating system.

Key Activation Example

The application's mapping file includes the following:

```
/* Menu Key */
    menu = pf1 (F1)
/* FRS command - print contents of screen */
    printscreen = controlG (Ctrl-G)
/* First item on menu line */
    menuitem1 = pf5 (F5)
/* Second item on menu line */
    menuitem2 = pf6 (F6)

/* Third item on menu line */
    menuitem3 = pf7 (F7)
/* Fourth item on menu line */
    menuitem4    = pf8 (F8)
/* Fifth item on menu line */
    menuitem5 = pf9 (F9)
/* Operation specified as FRS key 1 */
    frskey1 = controlK (Ctrl-K)
/* Operation specified as FRS key 2 */
    frskey2 = pf12 (Sh-F2)
```

embedded SQL (skeletal) program:

```
/* Operation code here */
exec frs display formname update;
exec frs initialize;
exec frs begin;
    /* Initialization operations here */
exec frs end;
exec frs activate frskey1;
exec frs begin;
        /* Operation code here */
exec frs end;

exec frs activate menuitem 'Help';
exec frs begin;
        /* Operation code here */
exec frs end;
exec frs activate menuitem 'Get';
exec frs begin;
        /* Operation code here */
exec frs end;

exec frs activate menuitem 'Add';
exec frs begin;
        /* Operation code here */
exec frs end;
exec frs activate menuitem 'End', frskey2;
exec frs begin;
        /* Operation code here */
exec frs end;
exec frs finalize;
```

The following menu line or key associations appear when a user runs the application:

```
Help(F5)  Get(F6)  Add(F7)  End(Sh-F2)
```

The menu items appear on the menu line in the order in which they were defined in the application. The label to the right of each menu item indicates the key to which it is mapped.

The first three menu items map to their respective keys by position. For example, the following statement in the application's mapping file maps the menu item in the first position on the menu line to F5:

```
menuitem1 = pf5 (F5)
```

This enables the end user to press F5 as an alternative to selecting the menu item Help by other means. The labels for the Get and Add menu items show a similar positional mapping.

The mapping file has a corresponding statement for the fourth menu item, which allows that item (End) to be invoked with F8:

```
menuitem4 = pf8 (F8)
```

The label for End in the menu display example indicates a mapping to Sh-F2 with the label (Sh-F2), rather than F8. To understand why, look back at the last operation defined in the application. The operation includes activations both for menu item End and for frskey2. Because of this, the user can invoke the operation either by pressing the key mapped to the End menu item by position, or by pressing the function or control key to which frskey2 has been mapped (in this case Sh-F2). The label for a menu item associated with a FRS key activation always indicates the FRS key mapping, rather than any mapping based on the menu item's position on the menu line.

In this example, the Menu key is F1. This was specified in the mapping file with the FRS command menu. The mapping file also mapped another FRS command, printscreen, to Ctrl-G. The printscreen command enables the user to send the contents of the screen to a file or a printer. Because this operation is implemented with a FRS command, it requires no coding within the application.

The first operation specified following initialization is not a menu item operation, but a FRS key operation for frskey1. The menu line indicates only operations that can be activated by menu items; therefore, the frskey1 operation does not appear in the menu. The only way a user can select this FRS key operation is by pressing the control or function key that has been mapped to it in the key mapping file. Because the example mapping file maps frskey1 to Ctrl-K, the user can invoke the operation with Ctrl-K.

The next example uses the same mapping file as above.

```
exec frs display formname another;
exec frs initialize;
exec frs begin;
    /* Initialization operations here */
exec frs end;
 exec frs activate menuitem 'Help', frskey1;
exec frs begin;

        /* Operation code here */
exec frs end;
 exec frs activate menuitem 'Find';
exec frs begin;
        /* Operation code here */
exec frs end;
 exec frs activate menuitem 'Add';
exec frs begin;

        /* Operation code here */
exec frs end;
 exec frs activate menuitem 'Delete';
exec frs begin;
        /* Operation code here */
exec frs end;
 exec frs activate menuitem 'Forget';
exec frs begin;

        /* Operation code here */
exec frs end;
 exec frs activate menuitem 'End', frskey2;
exec frs begin;
        /* Operation code here */
exec frs end;
 exec frs activate frskey3;
exec frs begin;
        /* Operation code here */
exec frs end;
 exec frs finalize;
```

This example generates the following menu line or key associations:

```
Help(CtrlK) Find(F6) Add(F7) Delete(F8) Forget(F9) End(Sh-F2)
```

The first menu item, Help, can be activated by a key mapped to a Forms Runtime System key, rather than to its position on the menu line. The next four items are invoked by keys associated with their positional mappings.

The End menu item has no positional mapping entry in the mapping file. Instead, the operation's definition includes an activation by frskey2, which is mapped to Sh-F2 in the mapping file. Because the menu item lacks a positional mapping, Sh-F2 is the only control key or function key that can be used to invoke the operation.

The final operation in the application specifies an activation by frskey3. Because the mapping file contains no mapping for frskey3, however, the user has no way of accessing this operation. Be sure that your mapping file includes entries for all FRS keys specified in your application.

Chapter 16: Table Fields

This section contains the following topics:

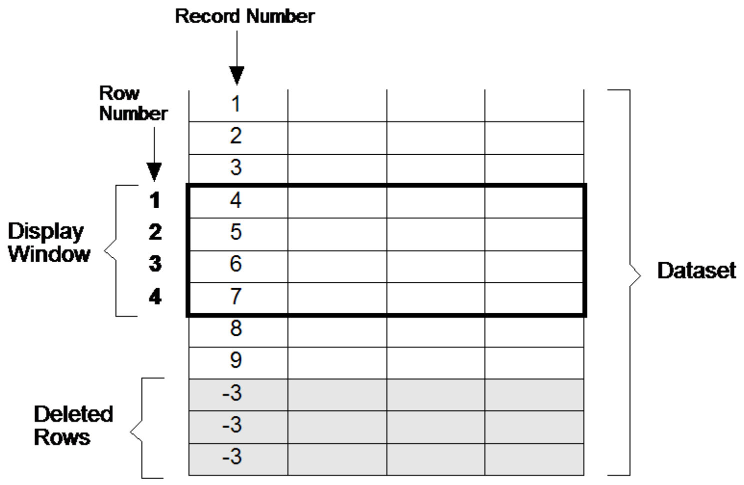
A *table field* is a form field that can display many rows and columns of data at a time. Each column is essentially a field in itself, and each row is a data record. Table fields provide a convenient means for the runtime user to view and edit large quantities of data. Table fields can be combined with simple fields—fields that hold only a single value—within a form. A form containing both table fields and simple fields is a natural choice for handling a master-detail type of application.

Overview of Table Fields

Table fields are defined in VIFRED and are manipulated by forms statements. Statements for handling table fields are similar to statements for handling simple fields. Among the operations you can perform with table fields are the following:

- Transfer data to and from columns in the table field
- Validate and clear rows of data
- Activate operations when the runtime user enters or leaves a specified table field column
- Return control to the FRS with the screen cursor on a specified column

The table field ordinarily acts as a window to an underlying set of data, usually data retrieved from the database. This *data set* can contain many more rows of data than can be displayed at one time in the table field, as shown in the following diagram:



Using forms statements, your application can allow the user to scroll to rows that are not initially displayed in the window. The user can add, delete or update rows in the data set, with the FRS tracking the changes until the changes are merged back into the database. An application can also manage table fields without any data sets, although in practice, this is uncommon.

Effective management of a table field—in particular, manipulation of its underlying data set—also requires additional forms statements unlike statements used to manage simple fields. These statements perform functions such as loading and unloading values into and out of the data set, inserting and deleting rows in the table field, and scrolling to rows in the data set.

The Data Set

In most instances, an internal data structure known as the data set is associated with the table field. The data set is linked to the table field when the table field is initialized. The data set stores all rows associated with the table field, whether or not the rows are displayed. Because the FRS manages the data set, a program need not be concerned with where data is coming from or going to during a scroll. The program can load many rows at once into the table field, even though the rows cannot be currently displayed. The program need not save deleted rows for later data manipulation because, although deleted rows disappear from the table field display, they still exist in the data set and can be used later for such tasks as updating the database.

How You Can Move Around a Table Field

The keystrokes that move the cursor between fields in a form also move the cursor between table field columns. Other keystrokes move the cursor up and down a row at a time.

When the user reaches the last row displayed by the table field, attempting to move down another row causes the next row in the data set, if any, to appear at the bottom of the table field. To accommodate the new row, the row at the top of the table field moves temporarily out of sight. This process is known as *scrolling*. Similarly, when the user attempts to move above the first displayed row of the table field, another scroll occurs, bringing into view the row in the data set that precedes the first displayed row and causing the last displayed row to drop out of sight.

For greater efficiency in scrolling, it is possible to scroll up and down a *page* at a time. A page corresponds to the number of rows that can be displayed at once in the table field. When the user scrolls up a page, the bottom row in the display becomes the first row, and the remaining rows in the display are taken from subsequent rows in the data set.

All of these movements are controlled by built-in functions (FRS commands) of the FRS, which are mapped to specific keys on your terminal. For a complete list of FRS commands and information on how to map them to your terminal, see the appropriate section of this guide.

In addition to the automatic scrolling provided by the FRS, you can define a scrolling operation that scrolls your display to a desired row in the data set. For information about these operations, see scroll Statement—Target Scrolls (see page 594).

Table Field Display Modes

Table fields have four display modes:

read

In this mode, the user can only browse the contents of the table field by scrolling through the values in the data set. This mode is normally used when a table field merely displays data or when all changes to the data are made by means of menu operations.

update

In this mode, the user can scroll through the values in the data set and change the data displayed. However, the user cannot add rows to the end of the data set. This mode is useful in applications that display a series of records for possible corrections by the user.

fill

This mode is similar to the update mode, except that it also allows the user to enter new rows at the end of the table field. Rows so entered are automatically recorded in the data set. Any column into which the user does not enter data is given a default value of zero or blanks or a null, depending on the column's data type and whether the column is nullable.

query

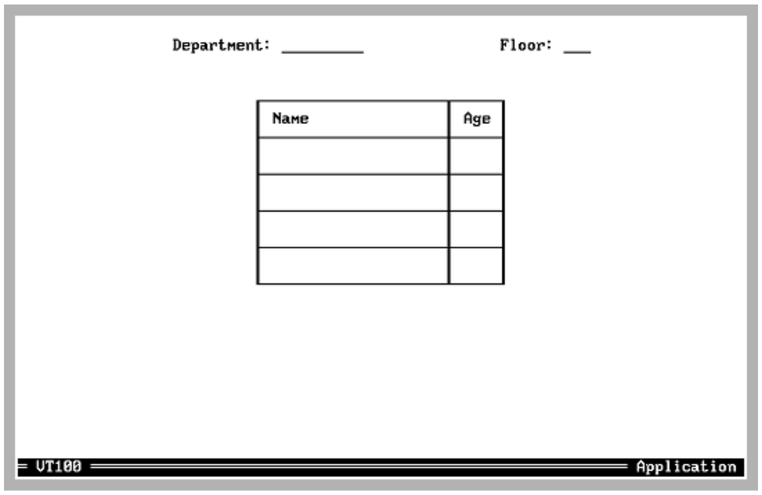
Like fill mode, new rows are appended when the user attempts to scroll beyond the end of the data set. In addition, query mode allows the user to enter query operators (for example, =) as part of the data for a column. This is useful for constructing a dynamic qualification for retrieval from the database.

The mode is defined when the table field is initialized. You can display the table field in any mode; however, if the form containing the table field is displayed in read mode, the table field itself also behaves as if it is in read mode despite its specified display mode.

For information about changing a table field's display mode while the table field is displayed, see How Table Field Modes Can Be Set Dynamically (see page 599).

A Sample Table Field

The examples in the remainder of this chapter all use the same form and database tables. The empform form, shown in the following figure, has one table field, employee, and two simple fields, department and floor. The table field has four displayed rows and two displayed columns, with the internal names of ename and age. The titles of the columns are Name and Age.



Department: ________ Floor: ___

Name	Age

Two database tables are used in the application, employee and department. They have the following descriptions:

```
exec sql declare employee table
     (eno   integer2  not null,
      ename  char(20)  not null,
      age    integer1  not null,
      sal    float4  not null,
      dept   char(10)  not null);
 exec sql declare department table
     (dept   char(10)  not null,
      floor   integer1  not null);
```

Also, it is assumed that the examples all contain the following host variable declarations:

```
exec sql begin declare section;
     eno        integer;
     ename      character_string(20);
     age        integer;
     sal        float;
     dept       character_string(10);
     floor      integer;
     state      integer;
     record     integer;
     msgbuf     character_string(80);
exec sql end declare section;
```

Where other variables are needed, their declarations are given in the example.

How Table Fields Can Be Used in an Application

There are three steps involved in managing a table field:

1. Initialize the table field using the inittable statement. This statement associates the table field with a data set.
2. Load values into the data set using the loadtable statement. After values are loaded and the form is displayed, the runtime user can browse and update the values. The loadtable statement loads values.
3. Unload values from the data set into program variables, using the unloadtable statement. The unloadtable statement sets up a loop that unloads values in the data set one row at a time, and executes a block of code for each row.

If your application starts with an empty data set, Step 2 is not required. If the table field is display-only, Step 3 is not required.

inittable Statement—Initialize a Table Field

A table field becomes associated with a data set when it is initialized. The initialization, which must occur after the form has been declared to the FRS, sets up the structure of the underlying data set; it does not actually put any data into the data set. The inittable statement initializes a table field..

This statement has the following syntax:

```
inittable formname tablename [tablemode]
    [(columnname = format {, columnname = format})];
```

formname

Specifies the name of the form in which the table field *tablename* is displayed

tablemode

Specifies one of the display modes described in Table Field Display Modes (see page 581) (read, update, query or fill). The default mode is fill.

You can optionally include *hidden* columns in the table field by enumerating them in the inittable statement. Hidden columns are identical to other table field columns, except that they are not displayed in the table field and therefore are invisible to the runtime user. The program can access values in hidden columns, just as it can access values in displayed columns. A typical use for a hidden column is as a repository for the unique identifying information (such as the employee number for an employee) of each row that is loaded into the data set from a database table. Once the user has modified the rows in the table field, the program can use the hidden column values within a search condition to update rows of the database table.

Unlike other columns in the table field, which are defined in Visual Forms Editor, hidden columns are defined when the table field is initialized.

columnname

Specifies the name by which the hidden column in the column list for inittable can be accessed

format

Indicates its storage format. The format must be a legal Ingres SQL data type from the following list.

Data Type	Meaning
char(1)-char(*n*)	Fixed length character string (all ASCII characters) of the specified length; *n* represents the lesser of the maximum configured row size and 32,000.

Data Type	Meaning
c1-c*n*	Fixed length character string (printing characters only) of the specified length; *n* represents the lesser of the maximum configured row size and 32,000.
varchar(1)-varchar(*n*)	Variable length character string (all ASCII characters) of the specified length; *n* represents the lesser of the maximum configured row size and 32,000.
text(1)-text(*n*)	Variable length character string (all ASCII characters except nulls); *n* represents the lesser of the maximum configured row size and 32,000.
integer1	1-byte integer
smallint	2-byte integer
integer	4-byte integer
float4	4-byte floating point
float	8-byte floating point
decimal	Exact numeric data type defined in terms of precision (total number of digits) and scale (number of digits to the right of the decimal point)
date	12-byte date value (displayed as a 25-character string)
money	8-byte money value (displayed as a 20-character string)

See the *SQL Reference Guide* for details about these data types.

The optional clauses with null and not null can follow the data type specification. By default, hidden columns are set to not null.

For example, the following embedded SQL statement sets up two hidden fields:

```
exec frs inittable empform employee update
     (sal = float4, eno = integer);
```

This defines a data set for the employee table field in the empform form. The table field is in update mode and contains two hidden columns, sal and eno. The table field also has the displayed columns defined for it in Visual Forms Editor; these are ename and age, with the titles Name and Age, as shown in the preceding figure.

loadtable Statement—Load a Data Set

To place data in a data set underlying an initialized table field, use the loadtable statement. Each execution of the loadtable statement adds one row of values to the data set. It is possible to load more rows than the table field can display. The user can access unseen rows by scrolling to them.

This statement has the following syntax:

```
loadtable formname tablename
 (columnname = data {, columnname = data});
```

formname

Specifies the name of the form in which the table field *tablename* is displayed

tablemode

Specifies one of the display modes described in Table Field Display Modes (see page 581) (read, update, query or fill). The default mode is fill.

columnname

Specifies the name of the table field column into which data is being loaded. It can be either a displayed or a hidden column.

data

Specifies the value, either a constant or a program variable, to be loaded into the column. Any column not specified is given a value of zero or blanks or a null, depending on its data type and whether the column is nullable.

Forms statements that refer to a column name must always refer to the column's *internal* name, as defined in VIFRED or as specified in the inittable statement for hidden columns. This name does not necessarily bear any relationship to the column's title, which appears above the column on the form.

By default, rows loaded into a table field using loadtable have a state of UNCHANGED. (See Table Field Row States (see page 601) in this chapter for more information about states.) You can, however, override this default when you load the table. This is done by specifying the *columnname* as the constant _state and assigning it a value corresponding to an alternate row state, such as UNDEFINED or NEW. See Setting Row States (see page 605).

The loadtable statement can occur at any point in the program after the table field has been initialized. If it is included either before or during the initialization of a display loop (see the following example), the values loaded appear when the form is first displayed. It can also appear inside an activate section, so that the table field is loaded as a result of the user selecting an operation.

This example initializes and loads a table field and its data set with values retrieved from a database table:

```
exec sql include sqlca;
 /* Data declarations listed earlier */
exec sql whenever sqlerror stop;
 exec sql connect personnel;
 exec sql declare emp_cursor cursor for
     select ename, age, sal, eno
     from employee
     where dept = :dept;
 exec frs forms;
exec frs forminit empform;


exec frs inittable empform employee update
     (sal = float4, eno = integer4);
 /* Get the department name and retrieve information */
exec frs prompt ('Enter department name: ', :dept);
exec sql select floor
     into :floor
     from department
     where dept = :dept;
 /* Display the form and initialize the fields */
exec frs display empform;


exec frs initialize (department = :dept, floor = :floor);
exec frs begin;
     exec sql open emp_cursor;
     exec sql whenever not found goto done;
    loop until no more rows
         exec sql fetch emp_cursor
              into :ename, :age, :sal, :eno;
         exec frs loadtable empform employee
              (ename = :ename, age = :age
              sal = :sal, eno = :eno);
         end loop;
 done:

exec sql close emp_cursor;
exec frs end;
 exec frs finalize;
 /* Update database table */
exec frs endforms;
exec sql disconnect;
```

This example prompts the user for a department name, fills the simple fields of the form with information on that department and then fills the table field with rows for the employees in the department. The loadtable statement is included as part of the initialization of the form's display loop; thus when the form appears, the table field already contains rows of values. Because the table field is in update mode, the runtime user can browse and update the rows in the table field. After the rows have been updated, the program must access the values in each row to use them to update the database table. This process is accomplished with the unloadtable statement.

unloadtable Statement—Unload the Data Set

The unloadtable statement provides a means by which the program can access values in each row of the data set in turn and perform actions based on those values.

This statement has the following syntax:

```
unloadtable formname tablename
   (variable = columnname {, variable = columnname});
 begin;
   program code;
 end;
```

When *tablename* is unloaded, the FRS first validates the values displayed in *tablename*. Then, the program starts with the first row in the data set and places the values in the specified *columnnames* into the *variables*. The host language and embedded query language statements in the statement block are then performed using those values. Next, the program unloads the second row in the data set and executes the statements on *those* values, and so on, through all the rows in the data set. The unloadtable statement causes a program loop to execute once for each row in the data set. It does not change any of the values in the data set; it merely provides a means for scanning the values. (In the unloadtable loop, avoid including any statements that change the ordering of the rows in the data set. Such statements can have unforeseen consequences.)

The unloadtable statement can appear wherever it makes sense to scan all the rows in the data set. A common location for it is at the end of a form's display, so that the values in the data set can be processed at that time. Be aware that an unloadtable loop does not clear the data set of its values. Data remains in the data set even after the end of a form's display. To clear the data set of values, you must use the clear field statement, with the name of the table field as the argument (see Ch. 17: Forms Statements (see page 613) for details).

You can terminate an unloadtable loop prematurely with the endloop statement. This statement breaks to the statement immediately following the unloadtable's program block. You can also terminate the unloadtable loop with a resume statement, back to the enclosing display loop.

During an unloadtable loop, the program can determine the record number and the state of the row being unloaded by using the special constants _record and _state as *columnnames* in the column list. These constants return integer values.

The _record constant returns the record number of the row currently being unloaded, with 1 signifying the first row in the data set. If the row has been deleted (see deleterow Statement—Delete Rows from the Table Field (see page 592)), _record returns a negative number. The _state constant returns an integer that corresponds to the row's state. The state indicates the most recent event a row's displayed columns have undergone.

Table Field Row States (see page 601) discusses the _state constant and its interaction with the various table field statements, and explains how to change table field row states dynamically.

Although a deleted row is no longer accessible through the table field display, with two exceptions, it still exists in the data set and can be accessed by an unloadtable statement. The exceptions are rows whose states were UNDEFINED or NEW before deletion. These rows disappear forever when they are deleted.

The next example shows a simple use of the _record and _state constants within an unloadtable loop and illustrates how a program can update the database by means of the changes made to the data set by the user.

```
/* Same declarations as in previous example. */
/* Initialize, load and display go here */
exec frs unloadtable empform employee
     (:ename = ename, :age = age, :sal = sal, :eno = eno,
     :state = _state);

exec frs begin;
     /*
     ** Based on the value of "state", the program can
     ** UPDATE, DELETE or INSERT values in the table
     ** "employee", using "eno" as the unique identifier
     ** in the search condition.
     ** In this example, the program only
     ** performs an UPDATE.
     */
```

```
    if (state = 3) then   /* State is CHANGED */
         exec sql update employee
              set ename = :ename, age = :age, sal = :sal
              where eno = :eno;
    end if;
exec frs end;
    ...
```

Table Field Operations

This section describes the various operations that you can perform on a table field and its underlying data set. These operations include inserting and deleting rows from a table field, transferring data to and from a table field, scrolling, and activation operations.

insertrow Statement—Insert Rows into the Table Field

The insertrow statement inserts a row at the beginning or middle of the data set.

This statement has the following syntax:

```
insertrow formname tablename [row]
    [(columnname = data {, columnname = data})];
```

Most of the syntax elements are described in loadtable Statement—Load a Data Set (see page 586). The *row* element, specifies where the new row is to be inserted. *Row* must be an integer value corresponding to the number of the *displayed* row in the table field after which the new row is to be inserted. *Row* must be in the range from 0 to the number of displayed rows as specified in Visual Forms Editor. If *row* is 0, the new row is inserted as the top row in the display. If *row* is omitted, the row is inserted after the row on which the cursor is currently positioned. In contrast to the constant _record, *row* refers to the row's position in the display, not to its position in the data set; rows added to the data set by insertrow always appear in the table field display.

Row cannot be greater than the number of rows actually containing data in the table field. For example, if the display has the capacity to hold four rows, but currently contains only two having data, then *row* must be in the range of 0 to 2, because the third and fourth rows do not yet exist.

When a new row is inserted, all rows below it are scrolled down. In a table field with four displayed rows, inserting a new row after row 2 causes row 3 to become the last displayed row and row 4 to be scrolled out of the display. When you insert a row after the last displayed row (in this case, by specifying *row* as 4), the rows in the display are scrolled up one, causing the first row to leave the display, and the new row appears as the last row in the display.

By default, rows inserted into a table field using insertrow have a state of UNCHANGED. (See Table Field Row States (see page 601) for more information about states.) You can, however, override this default when you load the table. This is done by specifying the *columnname* as the constant _state and assigning it a value corresponding to an alternate row state, such as UNDEFINED or NEW. See Setting Row States (see page 605) for more information.

The column list in insertrow is optional. If the column list is not included, the newly inserted row takes on the default value of zero or blank. By this means, your program can provide the user with the ability to add rows to the middle of the data set.

The following example demonstrates the insertrow statement. Assume that FRS key 5 has been mapped to Ctrl F. (FRS key mapping is described elsewhere in this guide.) Then, if you provide your program with the following activate section, a user can insert a row into the middle of a table field by pressing Ctrl F.

```
     ...
 exec frs activate frskey5;
exec frs begin;
     /*
     ** Insert a new row after the current row, leaving it
     ** clear for the user to fill.
     */
     exec frs insertrow empform employee;
exec frs end;
     ...
```

deleterow Statement—Delete Rows from the Table Field

When a row is deleted, it disappears from the table field's display. The remaining rows in the table field are compacted and a new row is scrolled in, so that a blank row is not left in the display. Even though the deleted row disappears from the display, normally it remains in the data set, with a state of DELETED. However, if the row's state before deletion was either UNDEFINED or NEW, then when you delete the row, it disappears forever; you cannot recover such a row.

To delete table field rows use the deleterow statement.

This statement has the following syntax:

deleterow *formname tablename* [*row*]

row

Specifies the number of the displayed row to be deleted. The specified row must be in the range from 1 to the number of rows in the table field display. If *row* is omitted, the current row is deleted.

Example:

```
...
 exec frs activate menuitem 'Delete';
exec frs begin;
     /* Delete the current row
     ** and save the employee number. */
     exec frs getrow (:eno = eno);
     exec frs deleterow empform employee
     /* Delete the corresponding row from the database. */
     exec sql delete from employee
          where eno = :eno;
exec frs end;
...
```

putrow Statement—Use the Program to Update Table Field Data

The putrow statement enables your application program to update values in an existing row. The putrow statement has the following syntax:

```
putrow formname tablename [row]
    (columnname = data {, columnname = data});
```

row

Specifies the number of the displayed row into which data is to be put. If *row* is omitted, the row on which the cursor currently sits is updated. If *row* is specified, it must identify a row currently displayed on the table field. You can use putrow inside an unloadtable loop. In that case, you must omit *row* because putrow updates the row currently being unloaded.

Only the *columnnames* specified receive new values; columns that are omitted are not changed. The value of *data* overwrites any value currently in the specified column.

getrow Statement—Get One Row of Data from a Table Field

The getrow statement transfers data from a single table field row to host variables.

This statement has the following syntax:

```
getrow formname tablename [row]
    (variable = columnname {, variable = columnname});
```

The getrow statement transfers values from table field columns for a specified displayed row into program variables. Getrow validates the row's values before assigning the values to the variables. If there is invalid data, an error message is displayed, the variables are not updated and execution continues with the next statement.

As in the unloadtable and deleterow statements, *columnname* can also be _state or _record.

This example below shows activate sections using the putrow and getrow statements.

```
     ...
 exec frs activate menuitem 'PutCurrent';
exec frs begin;
     /* Put new information into the current row. */
     exec frs putrow empform employee
          (age = :age, sal = :sal);
exec frs end;
 exec frs activate menuitem 'GetFirst';
exec frs begin;
     /* Get information from the first displayed row. */
     exec frs getrow empform employee 1
          (:age = age, :sal = sal, :state = _STATE);
exec frs end;
     ...
```

scroll Statement—Target Scrolls

To scroll to a specific record or to the top or bottom of the data set, use the scroll statement. These specialized scrolls are called *target scrolls*. A target scroll moves the cursor to a specified row in the data set.

This statement has the following syntax:

scroll *formname tablename* to *record*|**end**;

This statement moves the cursor to the row in the data set with the record number *record*. The key word end used in place of *record* causes the program to scroll to the last row in the data set. The following embedded SQL example contains three activate sections, each of which demonstrates a different use of the target scroll.

```
/* Assume previous declarations, plus ... */
exec sql begin declare section;
     searchname  character_string(20);
exec sql end declare section;
     ...
 exec frs activate menuitem 'Bottom';
exec frs begin;
     /* Scroll to end of data set */
     exec frs scroll empform employee to end;
exec frs end;
```

```
 exec frs activate menuitem 'Top';
exec frs begin;
     /* Scroll to first record in data set */
     exec frs scroll empform employee to 1;
exec frs end;
 exec frs activate menuitem 'Find';
exec frs begin;
     /* Prompt for name to search for */
     exec frs prompt ('Name to search for: ',:searchname);
     /* Loop through data set and stop
     ** when name is found.*/
     exec frs unloadtable empform employee
          (:ename = ename, :record = _record);

     exec frs begin;
          if (ename = searchname) then
              /* Scroll to record with specified name. */
              exec frs scroll empform employee to :record;
              exec frs resume field employee;
          end if;
     exec frs end;
     exec frs message 'Cannot find named employee.';
     exec frs sleep 2;
exec frs end;
     ...
```

In the Find operation above, when the name is found, the activate section is terminated by the resume statement. If the name has not been found by the time that all rows are unloaded, control passes to the message statement following the end of the unloadtable loop.

Another variant of the scroll statement functions identically to the automatic FRS scrolls, scrolling the displayed data set up or down one line at a time. This variant is used primarily in scroll activation blocks, which are described in the next section. For a discussion of all the scroll statement variants, see the scroll statement description in Forms Statements (see page 613).

Table Field Activation Operations

There are two types of activation operations that can be specified only for table fields: *scroll* activations and *column* activations. These are described in the following sections. For a discussion of activation operations in general, see Activation Operations (see page 551).

Scroll Activations

To define operations that are activated when the user attempts to scroll up or down the table field, use the activate scrollup and activate scrolldown statements. If these statements are included in a display block, automatic scrolling no longer occurs. Instead, when the user attempts to scroll in the specified direction, control passes to the activate block and the statements in that block are executed. If the block does not contain any statement that scrolls the table field, no scroll occurs. Therefore, unless your intent is to disable scrolling, you must include in such operations the variant of the scroll statement that functions like an automatic FRS scroll.

Situations in which you might want to activate an operation on a scroll attempt are extremely limited when dealing with table fields with attached data sets; ordinarily, the automatic FRS scrolls provide sufficient scrolling capability. One use for scroll activation might be to force a complex validation check (for instance, a check on a database table) on a row before it is scrolled. As mentioned above, you can also use scroll activations to disable scrolling. For instance, if you want to prevent the user from scrolling backward through the table field, you can include an activate scrolldown section with an empty statement block:

```
exec frs activate scrolldown employee;
 exec frs begin;
exec frs end;
```

An activate scrolldown is used here because scrolling backward through the form causes the rows above the table field to be moved down into the display.

An operation equivalent to automatic FRS scrolling can be specified as follows:

```
exec frs activate scrollup employee;
 exec frs begin;
     exec frs message 'Scrolling up';
     exec frs sleep 1;
     exec frs scroll empform employee up;
exec frs end;
```

Details on scroll activation can be found in the activate and scroll statement descriptions in Forms Statements (see page 613).

activate Statement—Activate on a Column

There are two types of column activations, *entry* and *exit*. If an entry activation is defined for a column, then activation occurs when the user moves the cursor into that column. Similarly, if an exit activation is defined for a column, then the activation occurs when the user moves the cursor out of a column.

The syntax for entry activation is as follows:

```
activate before column tablefieldname columnname|all;
 begin;
     program code;
 end;
```

The syntax for exit activation is as follows:

```
activate [after] column tablefieldname columnname|all;
 begin;
     program code;
 end;
```

If you specify the key word all in either statement, then the specified activation occurs on all columns in the table field.

A column activation can occur when the user moves the cursor into a column or out of the column, or both, if you supply two activate column blocks, one for each type of action.

A number of cursor movements can cause column activations: moving the cursor to the next or previous column, or moving the cursor up or down a row in the table field. Therefore, the FRS must remember what action caused the activate column statement, so that the action can be completed when the section of code has been executed. By using a resume next statement at the end of the code section, you can specify that the action that caused the activate column operation is to be completed after the operation finishes. In the following program fragment, the resume next, specifies that execution is to continue if the column contains good data; if the program detects bad data, it displays an error message and places the cursor in the column in which the bad data was entered by issuing the resume statement.

```
/* Assume previous declarations, plus ... */
exec sql begin declare section;
     rowcount integer;
exec sql end declare section;
     ...
 exec frs activate column employee ename;
exec frs begin;
     /*
     ** Get the name of the user and verify that it is
     ** unique in the Employee table. If it is, continue
     ** movement of cursor out of the Ename column; if it
     ** is not, remain in the same column.
     */

     exec frs getrow empform employee (:ename = ename);
     exec sql select count(*)
          into :rowcount
          from employee
          where employee.ename = :ename;
     if (rowcount = 0) then
          /* The name is unique, so continue. */
          exec frs resume next;
     else
          /* The name is not unique,
          ** so remain on column. */
          exec frs message 'Employee name must be unique';
          exec frs sleep 2;
          exec frs resume;
     end if;
exec frs end;
     ...
```

A program can position the cursor to a particular column in the current row using the resume column statement:

resume column *tablename columnname*;

How Table Field Modes Can Be Set Dynamically

To change the mode of a table field dynamically use the set_frs statement. The data set associated with the table field is retained unless the change is to or from the query mode. When the mode changes to or from the query mode, the data set is lost, as if the program had executed a clear field on the table field. In addition, changing a table field's mode can cause any derived fields dependent on the table field's columns to be marked invalid and recalculated. (For information about derived fields and their behavior, see Derived Fields (see page 608).)

The following table summarizes the behavior of the FRS when a table field mode is changed dynamically:

Starting Mode	**New Mode**			
	Fill	**Update**	**Read**	**Query**
Fill	—	Changes mode only; no other action	Validates table field	Clears data set
Update	Adds row	—	Validates table field	Clears data set
Read	Adds row	Changes mode only; no other action	—	Clears data set
Query	Clears data set	Clears data set	Clears data set	Clears data set

The short entries in the table indicate the following behavior:

Starting Mode	**New Mode**
Adds row	If the table field had no rows, changing to fill mode adds one row to the data set.
Clears data set	Clears the data set, including hidden fields, and moves the cursor to the first row and column of the table field. Performs any entry activations defined for that column if the cursor was not on that field before the mode change.
Validates table field	Validates the visible rows of the table field before changing the mode. If the validation fails, the mode is not changed.

The program can use the inquire_frs statement to determine the current mode of a table field if it is unknown.

For a complete discussion of the syntax of these statements, see the set_frs and inquire_frs statement descriptions in Forms Statements (see page 613).

Miscellaneous Statements

To sequence through all the columns in a table field, inquiring on each one, use the tabledata loop.

To validate table fields use the following statements:

validate

Validates all columns and rows of displayed data

validrow

Validates only the data in a single, specified row. Rows of data are automatically validated when they are scrolled out of the table field's display window. See How Field Validation Works (see page 609) for a full discussion of FRS validation operations.

To clear rows of data use the following statements:

clear

Removes all rows from a data set, including rows, which are not currently visible. Because the data set continues to store values even after the end of a form's display, the clear statement provides a convenient method for clearing out the table field display before redisplaying the form. The only other way to remove all values from the data set is to issue a second inittable statement before redisplaying the form, which causes a new, empty data set to overwrite the original one.

clearrow

Clears a single displayed row. Unlike the deleterow statement, which deletes the row from the display and compacts the remaining rows, clearrow does not delete the row, but merely clears its values.

For detailed descriptions of these statements, see Forms Statements (see page 613).

Table Field Row States

A table field displays a specified number of rows from its underlying data set. As with simple fields, you can edit the data displayed in table field rows. In addition, you can insert rows into the display and leave them blank or fill them with data. You can also delete a displayed row. The _state constant enables you to determine the status of a table field row.

The _state constant returns an integer value that corresponds to the row's state. In a conditional statement, placed within an unloadtable or getrow statement, its value can determine the disposition of the data in the row.

The following example of an unloadtable statement illustrates a typical use of _state:

```
exec frs unloadtable empform employee
     (:var1 = column1, :var2 = column2,
      :statevar = _state);
exec frs begin;
     /*
     ** Depending on the value of 'statevar,' perform an
     ** INSERT, DELETE or UPDATE to a database table,
     ** using the values unloaded from 'column1'
     ** and 'column2.'
     */
exec frs end;
```

Because it is intended to keep track of changes made by the runtime user, the _state constant only records changes made to the displayed columns of a table field, either visible or invisible. The _state constant does not record changes to hidden columns, because hidden columns are under the control of the application and are not affected by the runtime user. You can, however, record changes to hidden columns by initializing a special hidden column in the table field and changing its value when changes are made to other hidden columns by the program.

The _state constant can return the following values:

State	Value	Meaning
UNDEFINED	0	Empty row appended but not filled by runtime user
NEW	1	Appended and filled by user at run time
UNCHANGED	2	Loaded or inserted by program and still the same
CHANGED	3	Loaded or inserted by program, but has since been modified by user or program
DELETED	4	Loaded or inserted by program, but has since been deleted

The UNDEFINED Row State

This state signifies an empty row in a table field in fill mode. An UNDEFINED row results when:

- You move the screen cursor beyond the last row in the data set, which opens a new row, and then leave the row without putting any data into it.
- Neither the program nor the user enters any data into a table field following an inittable in fill mode. This results in one empty, UNDEFINED row.

Generally, UNDEFINED rows appear blank on the screen. However, the row can have zeros in numeric columns, if cursor movements caused a validation to occur. If an UNDEFINED row contains a derived value, that value is displayed if it is possible to calculate it.

The program must not try to use data in an UNDEFINED row.

The NEW Row State

A row in the NEW state contains newly added data. A NEW row was originally an UNDEFINED row into which data has been entered by the user or the program. If the user adds a new row by moving past the end of the data set and filling the newly opened row with data, the row has a state of NEW. This row remains in the NEW state even if a putrow statement overwrites the values the user entered. If a NEW row is deleted, however, it ceases to exist. It is important to distinguish between a NEW row added by the user and a row inserted into the table field with the insertrow statement. The latter row has a state of UNCHANGED, not NEW.

The UNCHANGED Row State

A row in the UNCHANGED state was either loaded or inserted into the table field by the program, using the loadtable or insertrow statement, respectively. The UNCHANGED state indicates that the data in the row has not been altered since the row was added. Any change to the data, caused either by the user typing over the row or by the program executing a putrow or clearrow statement on the row, sets the row's state to CHANGED. A blank row inserted into the table field by an insertrow statement has, as its original data, blanks and zeros or nulls, and, as its state, UNCHANGED. If the user types into the row the row's state becomes CHANGED.

The CHANGED Row State

A row in the CHANGED state was originally in the state UNCHANGED because either the user has typed into the row, modifying the original data, or the program has issued a putrow or clearrow statement, again modifying the data. In both of these cases, the row's state becomes CHANGED.

A blank row inserted by the program with an insertrow statement and then filled by the user attains the state CHANGED. Because the state CHANGED provides a useful way to indicate to the program that it must perform an update on a corresponding row in the database, programs that allow the user to insert a blank row (by executing an operation containing an insertrow statement) and fill it with data, must distinguish between a row originally loaded by the program and then changed (which must effect an update to the database) and a row originally blank and then filled by the user (which must effect an insert to the database). The simplest way to provide this capability is to initialize the table field with a hidden column. For all rows originally loaded into the table field from the database, this column can be given a value of 0. Whenever the insertrow statement adds a blank row to the table field, it can give the hidden column a value of 1 to distinguish it from the original rows.

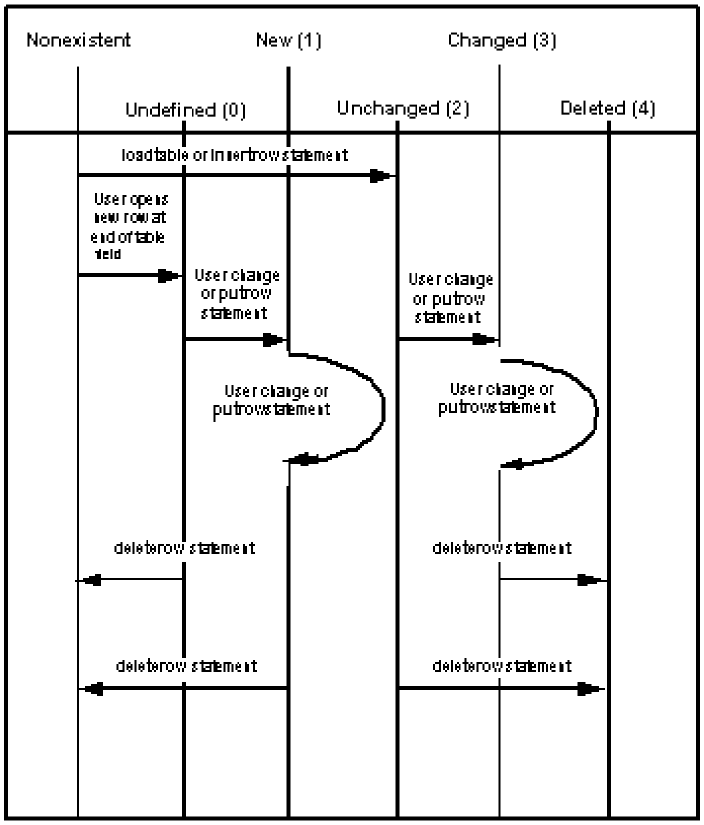
The DELETED Row State

An UNCHANGED row that has been deleted by a deleterow statement becomes a DELETED row. Even if the original data was changed sometime earlier (by the user or the program), causing the row's state to temporarily become CHANGED, upon deletion the row's state becomes DELETED, with no indication that the data was previously changed. By default, blank rows inserted using the insertrow statement, begin as UNCHANGED, not NEW, and therefore assume state DELETED if deleted. To assign a state other than UNCHANGED, use the insertrow(_state=*value*) statement; for details, see the insertrow statement description in Forms Statements (see page 613).

When you delete an UNDEFINED or NEW row, the row is physically removed from the data set. If you delete a CHANGED or UNCHANGED row, the row is removed from the display, but remains in the data set, and the values in the deleted row can be accessed, but not updated, using the unloadtable statement.

Effects of Forms Statements on Row States

The following figure summarizes the effects of forms statements and user actions on table field row states. The vertical lines represent the starting and ending row states, and the arrows are labeled to indicate the statement or action that causes the state to change. The footnotes for the diagram are on the page following the diagram.


Nonexistent
New (1)
Changed (3)
Undefined (0)
Unchanged (2)
Deleted (4)
loadtable or insertrow statement
User opens new row at end of table field
User change or putrow statement
User change or putrow statement
User change or putrow statement
User change or putrow statement
deleterow statement
deleterow statement
deleterow statement
deleterow statement

Note: After a program initializes a table field by issuing the inittable statement, the table field contains a single empty row (the fake row), with an UNDEFINED state (assuming the table field mode is fill). The program detects the fake row if the program issues an inittable statement followed immediately by an unloadtable statement, or issues the inquire_frs table...(datarows) statement.

The user can open a new row at the end of a table field only if the table field mode is fill or query.

Table Field Modes and Row States

The following table summarizes the row states that can occur for each table field mode.

	Undefined	**New**	**Changed**	**Deleted**
Fill	yes	yes	yes	yes
Update	no	no	yes	yes
Read	no	no	yes	yes
Query	yes	yes	yes	yes

The row is marked CHANGED if the application program executes a putrow or clearrow statement that changes data in the row.

Setting Row States

When a row is placed into the data set, using either the loadtable or insertrow statement, by default, that row's state is UNCHANGED. To set the row state differently, specify the _state constant in the argument list of the statement, assigning it the desired state. You can also use the putrow statement to change a row's state (note that only certain states are allowed, based on the current state — for more information on the putrow statement, see Forms Statements (see page 613)). You can specify _state as an integer literal or an integer variable.

For example, the following insertrow statement inserts a row into the table field and assigns it a NEW state:

```
exec frs insertrow deptform emp_table
     (ename = :name, esalary = :salary, _state = 1);
```

You can specify any legal _state value except DELETED. The legal values and their corresponding states are:

0—UNDEFINED
1—NEW
2—UNCHANGED
3—CHANGED

If you specify an UNDEFINED state, do not specify values for any visible columns in the row, because, by definition, an UNDEFINED row is a row that is empty. If you try to do so, you receive a runtime warning but Ingres assigns the values to the visible columns. Ingres generates the warning because the existence of data in an UNDEFINED row can cause problems when you use _state to decide which type of database update to perform during an unloadtable or getrow. You can, however, assign values to hidden columns regardless of the state assigned to the row.

Note: Derived values in UNDEFINED rows are displayed if it is possible to calculate them.

Row States and Floating Point Data

If the display format of a table field column is not large enough to accommodate a floating point value, the value is rounded for display. As a result, the value displayed is different from the value in the table field data set, and the row's state is set to CHANGED (only if the user types into field). To avoid this, allow enough precision in the format for the floating point data as stored in the database table.

Change Variables

The FRS maintains a change variable, set by the user typing in a field, for each form and each field on a form. In addition, if the field is a table field, each element in the table field's data set, except hidden columns, also has an associated change variable.

The change variable for the form is set whenever the user makes any change to any field on the form. Similarly, a change variable associated with a specific field or a data set element is set whenever the user types into that field or changes that element. This change variable is cleared whenever a value is placed into the field by the program (by way of the clearrow, putrow, loadtable, and insertrow statements). However, a change in the value of a derived field does not set or clear the change variable for the derived field.

Form and field change variables are maintained separately; for example, if you clear all field change variables, Ingres does not automatically clear the form change variable—your application program must clear the form change variable separately.

The FRS scrolls the change variables when the table field scrolls.

A field change variable is useful for determining which fields on a form or in a table field have been changed by the user. An application can thus optimize special data checks by checking only the elements that have actually changed.

You can set the change variable or obtain its value by using the set_frs and inquire_frs statements. For example, to set a table field change variable, the syntax is:

```
exec frs set_frs row 'formname ' 'tablefieldname ' [row#]
 (change (columnname) = integervar)
```

To obtain the value of a table field change variable, the syntax is:

```
exec frs inquire_frs row 'formname ' 'tablefieldname ' [row#]
 (change(columnname ) = integervar )
```

Invisible Fields

One of the attributes that you can set for simple fields, table fields, and table field columns is invisibility. An invisible field or column does not appear on the form when the FRS displays the form. You can set this attribute when the form is defined in VIFRED or at run time, using the set_frs statement.

When you make a table field column invisible, all columns that normally appear to the right of the invisible column are shifted left so that no blank space appears in the table field display.

To make an entire table field invisible, do one of the following:

- Make the table field itself invisible
- Make all the columns in the table field invisible

If you make a table field invisible, then the attribute settings for its component columns are ignored; the entire table field is invisible even if some of its columns are still set to visible.

As with other attribute settings, the application can use the inquire_frs statement to determine the current setting of the invisibility attribute. For a complete description of the syntax of both the inquire_frs and the set_frs statements, see Forms Statements (see page 613).

Derived Fields

A *derived* field is a simple field or table field column whose value is derived from the value of one or more fields or columns on the same form or a constant. The value is determined by a formula that is specified when the form is defined in VIFRED. (See the appropriate section of this guide for information about defining a derived field.)

The derivation formulas for simple fields can use constant values, values in other simple fields, or aggregate values of non-hidden table field columns. The formulas for table field columns can use constant values or values in non-hidden columns in the same table field row.

A field or column that is directly referenced in another field's derivation formula is said to be a direct source field for that field. For example, if field A is dependent on the values in fields B and C, then fields B and C are source fields for field A. A source field can itself be a derived field. For example, if field C is itself dependent on fields F and G, then fields F and G are the direct source fields of field C and indirect source fields of field A.

You can nest field dependencies to any level. However, you cannot define circular dependencies. That is, you cannot derive field A from field B, which depends on field C, if field C is derived from field A. Circular dependencies generate an error, either when the form is saved or when the form is specified with the addform or forminit statement in an application. When this error occurs, the form is not displayed.

Values in derived fields are valid only when the values in all source fields, direct and indirect, are valid, with one exception. The exception occurs in the context of table field aggregates. Invalid values in a table field column are ignored during aggregate processing, which means that an aggregate value of a table field column is never invalid. (See the appropriate section of this guide for a complete description of how aggregates are used in derivation formulas.)

Because source fields must be valid, the FRS calculates derived values after the source fields are validated. If the user enters data into the source fields, the validations occur according to the tables shown in How Field Validation Works (see page 609). If user-entered data is not valid, then no derivation calculations are performed and derived fields are blanked out. When the program enters data into the source fields, the validation occurs immediately after data entry.

The FRS also calculates derived field values, if possible, whenever a user creates a new row by scrolling the cursor past the end of the data set or whenever an insertrow or loadtable creates a new row. Any derived value in an UNDEFINED table field row is displayed if it can be calculated. However, a derived simple field or table field column is not displayed if the form or table field, respectively, is in query mode.

Whenever you or the program attempts to retrieve a derived value, through a getform, getrow, unloadtable, or finalize statement, the FRS checks the validity of all source fields of the derived value. If a source field has been changed, it is validated and the derived value is recalculated. If the source field fails the validation check, then the derived value is marked invalid and it is not placed into the designated variable.

The FRS marks derived values invalid and clears the field whenever:

- A source field contains an invalid value.
- A source field is cleared.
- An exception, such as math underflow or overflow or value truncation or rounding, occurs during a derivation calculation.

  Some environments do not signal floating division by zero as an exception.
- A form is displayed in query mode.
- A source table field is in query mode.
- A table field containing a column that is a derived field is in query mode.

  Even if the derived field (column) is derived from constants, it is still cleared if its parent table field is in query mode.
- The form's mode is changed to query mode.
- An explicit validation of a derived field fails.

  An explicit validation fails if any of the source fields fails its validation.

Because values in derived fields are determined by a predefined formula, it is not possible for either the program or the user to access these fields to change or edit their values. Because you cannot move the cursor into a derived field, you cannot define a field activation on a derived field. If you try to do so, you receive a runtime error.

How Field Validation Works

Validation is a process performed by the FRS, using criteria specified in a form's VIFRED definition, to ensure that data in a field is valid. Validation checks can be performed on all fields on a form or on individual fields. When the validation is performed on a table field, the validation can affect either a single cell in the table field or an entire row. Validation checks can occur when the user moves the cursor out of a field, selects a menu item, presses the menu key, presses a function or control key mapped to an FRS key, or scrolls a table field. They can also occur if the program issues either a validate or validrow statement.

There are three steps to the field validation process:

1. If the field was defined in VIFRED with the mandatory attribute, the FRS verifies that a value is present in the field.
2. If the field has a numeric, date, or money format, the FRS verifies that the data in the field is the correct type.
3. If the field was defined in VIFRED with a validation check, the FRS checks that the field's entry matches the criteria specified.

When the validation is performed on a table field row, all non-hidden and non-derived columns, including display-only columns, are validated.

If the validation check fails at any step in the process, the current operation stops and the Forms Runtime Editor returns the cursor to the beginning of the field that failed the check.

Note: When the validation occurs as a result of a getform, getrow, putform, or putrow statement, the program continues with the following statement even if a field fails the validation check.

There are several methods to specify when validation checks are performed in an application. The set_frs frs statement, using the validate constant, can be used to set global validation defaults for forward tabbing, backward tabbing, menu items, frskeys, and the menu key. You can override the default settings for specific menu items and frskeys with the activate statement (with the *menuitem* and *frskey* conditions). Or, you can issue a validate or validrow statement inside an initialize or activate block.

The display mode of the form or table field also affects whether or not a validation occurs.

The following tables describe the effects of form modes on field validations. The tables indicate whether validations are performed; a Y means that all three steps of the validation are performed, an N means that a validation is not performed, and a D indicates that only Step 2 is performed. The numbers after some entries refer to the footnotes following the tables.

IN SIMPLE FIELDS	**Form Display Mode**			
Action or Statement	**Fill**	**Read**	**Update**	**Query**
User moves cursor forward out of the current field to new field. Is the current field validated? 1	Y	N	Y	N
User moves cursor backward out the current field to a new field. Is the current field validated? 1	N	N	N	N
The getform statement references a simple field. Is the field validated?	Y	Y	Y	D

IN SIMPLE FIELDS	**Form Display Mode**			
Action or Statement	**Fill**	**Read**	**Update**	**Query**
The putform statement references a simple field. Is the field validated?	D	D	D	D
Program performs a validate fieldname against a simple field. Is the field validated?	Y	Y	Y	D

ACTIONS NOT DEPENDENT ON FIELD TYPE	**Display Mode**			
Action or Statement	**Fill**	**Read**	**Update**	**Query**
User presses the menu key or a control/function key mapped to a FRS key. Is the current field validated? 1	N	N	N	N
User presses the menu key or a control/function key mapped to a FRS key. Validation checking is on for that key. Is the current field or table field column validated?	Y	N	Y	D
The program issues an enddisplay. Are simple fields and the visible table field rows validated?	Y	Y	Y	D
The program issues a breakdisplay. Are simple fields and visible table field columns validated?	N	N	N	N

1. These are the default validations. You can override these validations with the set_frs or activate statement.
2. You cannot use the set_frs statement to override these validations.

Chapter 17: Forms Statements

This section contains the following topics:

This chapter describes forms statements. The general syntax for each of these statements is:

```
[margin] exec frs | ## forms_statement [terminator]
```

Because the use of a *margin* and *terminator* are dependent on your host language, you must consult your host language companion guide for information about these portions of the syntax.

If you are programming in embedded SQL, you must precede forms statements with exec frs; if you are programming in embedded QUEL, you must precede forms statements with **##**. The statement syntaxes in this chapter omit these syntax elements. The examples in the chapters in this part use SQL syntax; for examples, QUEL users should see EQUEL/FORMS Examples (see page 757).

Because the statement terminator is host-language dependent, the formal syntax presented in this chapter does not contain a statement terminator; the examples in each section use a semicolon (;) as a statement terminator.

Many statements require you to specify a field name or column name, or both. In this case you must specify the internal name of the field or table field column as specified in VIFRED. This internal name does not necessarily bear any relationship to the title of the field or column, although the title is displayed with the field on the form.

activate Statement—Activate a Section of Code

This statement activates a section of program code within a display block.

Syntax

```
activate condition [objectname]
     [([validate = value]
     [, activate = value])]
     {, condition [objectname]
     [([validate = value]
     [, activate = value])]}
 begin
      program code;
 end
```

Description

When the specified condition occurs, the activate statement interrupts the form's display loop and executes the associated block of code. Each condition represents a user action. When the user performs the indicated action, the associated block of code is executed. After the code is executed, control returns to the FRS and the display loop continues (unless the code executed a statement that terminated the loop).

You must place the activate statement in a display block. (A display block is that portion of code in a forms-based program that displays a form. For a discussion of display blocks and the structure of a forms-based program, see Embedded Forms Program Structure (see page 541).)

The begin/end block contains the code that is executed when the condition is true. You cannot put any other statements or any comment lines between the activate statement and its associated begin/end block, or between activate sections. Any such statements or comments generate a preprocessor error.

Condition can be any of the following:

- **menuitem**
- **before field** or [**after**] **field**
- **before column** or [**after**] **column**
- **scrollup** or **scrolldown**
- **frskey***N* or **frskey** *N* or **frskey** *integer_variable*

  (The space between frskey and *N* is optional.)
- **timeout**

The exact syntax of the activate statement varies with the chosen condition. The subsections of this statement description describe each of these conditions and its appropriate syntax. You can specify more than one condition in a single activate statement (see the examples at the end of this description).

Objectname identifies the particular object on which the specified *condition* is performed. For example, if the *condition* is menuitem, then *objectname* identifies a particular menu item and whenever the user selects that menu item, the associated block of code executes. Not all of the conditions require an *objectname*; see the separate discussions of each condition for details.

When you specify the menuitem and frskey conditions, you can include the validate and activate clauses in the statement.

The validate clause causes a validation check on the current field to take place when the activate statement is executed. (Validation checks on fields are described in Table Fields (see page 579).) The activate clause causes any activate field or activate column statement associated with the current field or column to be executed before executing the activate statement associated with the menuitem or frskey. These clauses allow you to build checks and controls into the application. For example, you can use these clauses to ensure that a particular field has a valid entry before allowing the user to leave the field. An example at the end of this section illustrates using the activate clause in conjunction with field or column conditions.

The *value* must be either 1 or 0, or an integer variable evaluating to 1 or 0. If *value* is 1 for the validate clause, the FRS validates the current field before the section of code is executed. If the field fails the check, an error message is displayed, the cursor is positioned back on the field, and the section of code is not executed. When the activate clause has a value of 1, the code associated with the field or column condition for the current field or column is executed before the code for the selected menuitem or frskey condition. Assigning a value of 0 to either the validate or activate clause turns off (overrides) any validations or activations that were defined for the specified condition using the set_frs statement.

If you include both the validate and activate clauses, you must separate them with a comma.

You can always include an explicit validate statement in the associated block of code, regardless of whether the activate statement contains a validate clause.

The Menuitem Condition

The menuitem condition has the following syntax:

```
activate menuitem menuname
    [([validate = value] [, activate = value])]
begin
    program code;
end
```

When you specify the menuitem condition, the associated program code is executed when the user selects the menu item identified by *menuname*. *Menuname* can be a quoted or unquoted character string or a program variable. If the *menuname* is a reserved word, you must use quotes. Its length cannot exceed the width of the screen minus 15 characters for a user input area. When the form is displayed, *menuname* appears on the menu line beneath the form.

If the display block contains any activate menuitem or activate frskey statements, you must provide an explicit means for the user to exit from the display loop (for example, a menu item operation containing an enddisplay or a breakdisplay statement). Otherwise, the user is not able to end the form's display. (For more information about terminating a form's display, see Embedded Forms Program Structure (see page 541).)

The Field Condition

For the field condition you can specify two options:

Entry activation

Specifies that the code is executed when the user moves the cursor into the specified field

Exit activation

Specifies that the code is executed when the user moves the cursor out of the specified field. (Field exit activations do not occur if the form is displayed in read mode.)

To specify an entry activation, use the following syntax:

```
activate before field fieldname | all
begin
     program code;
end
```

To specify an exit activation, use the following syntax:

```
activate [after] field fieldname | all
begin
    program code;
end
```

By default, the activation occurs only when the user moves the cursor into or out of a field. (Entry activations are also executed when the resume field statement places the cursor on a field.) If you want to execute the code as a result of some other user action, for example, whenever the user presses a function key or selects a menu item while the cursor resides on the specified field, use the set_frs statement or an explicit activate clause. The set_frs statement, described later in this chapter, can define global field activations. See the descriptions of the menuitem and frskey conditions for a description of how to use the activate clause.

Fieldname must be a quoted or unquoted character string or program variable identifying a simple field. If you specify all, the program code is executed upon entry or exit, as specified, of all simple fields on the form. (To specify entry or exit activation of table field columns, use the activate column version of the activate statement.) *Fieldname* cannot identify a derived field, because a user cannot move the cursor to these fields.

If a display block includes an activate field all and an activate section for an individual field, only the code associated with whichever of the two activations appears second in the display block is executed when an activation on that particular field occurs.

By default, after an exit activation, the FRS positions the cursor back on the field that caused the activation. Consequently, be sure to include a resume statement that moves the cursor to another field when the operation is complete.

The resume next statement is very important if the field activation is the result of a menuitem or frskey selection. Without this statement, the block of code associated with the selected menuitem or frskey condition is not executed. It appears as if the menuitem or frskey condition was never selected.

The Column Condition

For the column condition you can specify two options:

Entry activation

Specifies that the code is executed when the user moves the cursor into the specified table field column. Entry activations are also executed when the resume column places the cursor in a column.

Exit activation

Specifies that the code is executed when the user moves the cursor out of the specified table field column. (Exit activations do not occur if the table field is displayed in read mode.)

To specify an entry activation, use the following syntax:

```
activate before column tablename columnname|all
begin
    program code;
end
```

To specify an exit activation, use the following syntax:

```
activate [after] column tablename columnname|all
begin
    program code;
end
```

By default, the activation occurs only when the user moves the cursor into or out of a column (depending on which variation you chose). When you specify an exit activation, moving the cursor up or down a row in the table field display also constitutes an exit and executes the program code.

If you want to execute the code as a result of some other user action, for example, whenever the user presses a function key or selects a menu item while the cursor resides in the specified column, use the set_frs statement or an explicit activate clause. The set_frs statement (see page 724) can define global column activations. See The Menuitem Condition (see page 617) and The Frskey Condition (see page 621) for a description of how to use the activate clause.

You can specify both *tablename* and *columnname* as quoted or unquoted character strings or as program variables. If you use the key word all instead of a specific column name, then the program code is executed whenever any column in the table field is entered (or exited, depending on what you specified).

By default, after an exit activation, the FRS positions the cursor back on the same column. For this reason be sure to include a resume statement that moves the cursor to another field or column when the operation is complete.

The resume next statement is very important if the column activation is the result of a menuitem or frskey selection. Without this statement, the block of code associated with the selected menuitem or frskey condition is *not* executed. It appears as if the menuitem or frskey condition was never selected.

Because it is possible for the user to initiate a scroll upon leaving a column (for instance, by attempting to move the cursor down a line when it is already on the last row of the table field), it is important to understand what happens when the display block includes activate column and activate scroll statements, both of which are affected by the user's action. In such a case, the activate column statement has the higher priority and is executed first. If execution of the activate column terminates with a resume next statement, the program next processes the activate scroll statement.

The Scroll Condition

The **scroll** condition has the following syntax:

```
activate scrollup | scrolldown tablename
begin
    program code;
end
```

The scroll condition is activated when the user attempts a scroll in the table field specified by *tablename*.

The scrollup condition occurs when the user tries to move the screen cursor past the last displayed row of the table field. This activates the scrollup condition, because the rows in the table field must move up to bring in any rows below the display. The scrolldown condition occurs when the user tries to move the cursor above the first row of the table field; the existing rows in the table field must move down to accommodate the new row coming in at the top.

In a table field the user can also activate scrolling by attempting a page scroll up or down using an FRS command. (FRS commands are built-in functions that control cursor movement and screen display. See the *Character-based Querying and Reporting Tools User Guide* for more information about them.) When a page scroll is attempted, the activate scroll block is executed for each row that the cursor scrolls through. Execution of the scroll statement does not constitute a scroll attempt and does not activate the scroll condition.

Ordinarily a scroll attempt automatically causes the specified scroll to occur. However, if an activate scroll is specified for the particular direction, that automatic scrolling does not occur. Instead, the program executes the statements in the activate block. A scroll statement must be included in the statement block to actually effect the scrolling. If the block does not include a scroll statement, the user is not able to scroll in the direction specified in the activate statement. A simple way to disable scrolling in a particular direction is to include an activate scroll statement with an empty program block for that direction.

Applications that use table fields do not commonly contain activate scroll blocks, except to handle certain special circumstances. For instance, it might be necessary to force a complex validation check, such as a check using a database table, on a row before it is scrolled.

The *tablename* is the name of a currently displayed table field. It can be specified as a character string, with or without quotes, or as a program variable.

The Frskey Condition

The **frskey** condition has the following syntax:

```
activate frskeyN|frskey keynumber
     [([validate = value] [, activate = value])]
begin
    program code;
end
```

The frskey condition causes the accompanying section of code to be executed when the user presses the control, arrow, or function key mapped to the designated FRS key. *Keynumber* can be an integer literal or integer variable. The number specified by *N* or *keynumber* must be from 1 to 40.

The *Character-based Querying and Reporting Tools User Guide* explains FRS key mapping in detail.

The Timeout Condition

The **timeout** condition has the following syntax:

```
activate timeout
begin
    program code
end
```

This condition is activated whenever the display loop times out. A display loop times out when the user fails to perform some keyboard action, such as moving the cursor, entering data, or making a menu choice, before a specified time period expires. (For a general discussion of the timeout feature and to the set_frs statement description for information about setting the timeout period, see Embedded Forms Program Structure (see page 541).)

When an activate timeout block that is part of a display or display submenu loop completes or executes a resume statement, the cursor returns to the current field or column. If it cannot return to the field or column, it returns to the menu line.

If the block is part of a submenu statement, execution of a resume or completion of the block results in completion of the submenu statement.

Examples—activate statement:

Example 1:

Create a Help menu item.

```
exec frs activate menuitem 'Help';
exec frs begin;
    exec frs help_frs (subject = 'Application',
      file = 'application.hlp');
exec frs end;
```

Example 2:

Cause execution of an operation when the user leaves the field specified by the variable fieldvar. If there were no errors, then continue with the user's movement out of the field. Otherwise, the default operation — resuming on the same field — is executed.

```
exec frs activate field :fieldvar;
exec frs begin;
    if (no error) then
         exec frs resume next;
    end if;
exec frs end;
```

Example 3:

Cause execution of an operation when the user either selects the Next menu item or presses the function/control key mapped to FRS Key 3; validate in either case.

```
exec frs activate menuitem 'Next' (validate=1),
    frskey3 (validate=1);
exec frs begin;
    exec frs fetch cursor1
         into :ename, :age;
    exec frs putform empform
         (ename = :ename, age = :age);
exec frs end;
```

Example 4:

When the user moves out of field key, provide help if the value of the field is ?. Otherwise, validate the value.

```
exec frs activate field key;
 exec frs begin;
    exec frs getform keyform (:key = key);
    if (key = '?') then
        exec frs help_frs
           (subject = 'Keys', file = 'keys.hlp');
        exec frs resume;
    end if;
    found = 0;
    exec sql select 1
        into :found
        from keys
        where key = :key;
    if (found = 1) then
        exec frs resume next;
    else
        exec frs message
           'Unknown key, please modify.';
        exec frs sleep 2;
    end if;
    /* Default action is to resume on same field */
exec frs end;
```

Example 5:

Prevent the user from scrolling backward through the employee table field, by disabling the scrolldown operation.

```
exec frs activate scrolldown employee;
 exec frs begin;
exec frs end;
```

Example 6:

Initiate a database validation check on the ename column before scrolling to the next row in the employee table field.

```
exec frs activate scrollup employee;
 exec frs begin;
    exec frs getrow empform employee
        (:ename = ename);
    exec sql select count(*)
        into :rowcount
        from employee
        where employee.ename = :ename;
    if rowcount = 0 then
        exec frs message
            'The employee entered does not exist';
        exec frs sleep 2;
    else
        exec frs scroll empform employee up;
    end if;
exec frs end;
```

Example 7:

When the user attempts to move the cursor out of the sal column, the activate column block is executed. If the value in sal is valid, the program continues with whatever action initiated the activate column block. If the user had been attempting to move the cursor down a row while on the last displayed row of the table field, the activate scrollup block is executed at this time. If, however, the value in sal is not valid, the cursor resumes on the original column and row, and the activate scrollup block is not executed.

```
exec frs activate column employee sal;
 exec frs begin;
    if sal < 40000.00 then
        exec frs resume next;
    end if;
    exec frs message 'Reduce salary';
    exec frs sleep 2;
    /* By default, the screen cursor resumes
    ** on the same column. */
exec frs end;
 exec frs activate scrollup employee;
exec frs begin;
    program validation code;
    exec frs scroll empform employee up;
exec frs end;
```

Example 8:

This example shows how the activate clause can be used to simplify data checking. A given form has only one field, the emp_name field, which must contain a valid name before the user is allowed to leave the field or select either menu item, run or frskey3.

```
exec frs activate field emp_name;
 exec frs begin;
    check that emp_name contains a value;
    if fail then
         exec frs resume;
    end if;
    exec frs resume next;
exec frs end;

 exec frs activate menuitem 'Run' (activate = 1);
exec frs begin;
    do processing based on value in field emp_name;
 exec frs end;
 exec frs activate frskey3 (activate = 1);
exec frs begin;
    give raise to employee named in field emp_name;
 exec frs end;
```

addform Statement—Declare a Pre-compiled Form

This statement declares a pre-compiled form to the FRS.

Syntax

```
addform :formname
```

Description

The addform statement declares a pre-compiled form to the FRS. Declaring a form makes its definition known to the FRS. A pre-compiled form is a form that has been compiled into object code. You cannot use the addform statement to declare a form that resides in a database. For these forms, you must use the forminit statement instead. All forms must be declared with either addform or forminit before you can use them in the application. (It is only necessary to declare a form once in an application, regardless of how many times it is displayed.)

Pre-compiling a form eliminates the need to access the forms catalogs in the database for the form's definition. This substantially reduces the time required to start a forms application and also allows you to use the FRS on the form without necessarily being connected to any database. However, because the form is pre-compiled, any changes made to the form through VIFRED necessitate its recompiling and relinking into the application.

Before you can issue this statement, you must:

1. Create and compile the form in VIFRED.
2. Compile the compiled form into object code.
3. Link the form object code into the program.
4. Declare a program variable for the address of the form's definition.

See your host language companion guide and *Character-based Querying and Reporting Tools User Guide* for complete information about these procedures.

The *formname* is the name given the form in VIFRED. Some implementations require that *formname* be a variable or string literal specifying a file name. See your host language companion guide for details.

You must include the colon in front of the *formname* because addform references the variable that contains the form's address, not the form name itself. (This is the only time in the program that *formname* is preceded by a colon.)

If the specified form has a VIFRED table look up validation defined for it of the type field in table.column, then a run time query against the database is issued when the addform executes. That query reads all the values in table.column into memory. If no transaction is open when this occurs, then addform issues a commit afterwards, which releases all its locks on the table from which the validation data is read. If a transaction is open, then addform does not commit afterwards. The application continues to hold the shared locks until the next commit or end transaction statement. These locks allow other users to select from the tables but prevent anyone from changing them. Because of this, applications must, whenever possible, commit open transactions before issuing the addform statement.

Examples—addform statement:

Example 1:

Declare the form empform.

```
exec sql begin declare section;
    empform external integer;
exec sql end declare section;
    ...
 exec frs addform :empform;
```

Example 2:

Inside a local procedure, which is called more than once, you control the reissuing of the **addform** statement.

```
exec sql begin declare section;
    empform external integer;
    added integer; /* statically initialized to 0 */
exec sql end declare section;
...
 if (added = 0) then
    exec frs addform :empform;
    added = 1;
end if;
```

breakdisplay statement—Terminate a Display Loop or Submenu

This statement terminates a display loop or a display submenu without performing field validations or executing the finalize statement.

Syntax

```
breakdisplay
```

Description

The breakdisplay statement terminates a display loop, returning control to the first statement in the program following the end of the display block. (For information about the structure of display blocks, see Embedded Forms Program Structure (see page 541).) Unlike the enddisplay statement, breakdisplay does not perform a validation check on the fields in the form nor does it execute the finalize statement.

The breakdisplay is particularly useful if you do not want to retain the data in the form, because validation checking is unnecessary in such instances.

The breakdisplay statement must be syntactically within the scope of a display block, as it generates a local goto statement to the end of the block. If the breakdisplay statement is inside an unloadtable statement which is itself nested in an activate section, breakdisplay exits the unloadtable loop, in addition to terminating the display loop. The second example demonstrates this.

Examples—breakdisplay statement:

Example 1:

The Quit operation terminates display of the form without validation checking.

```
exec frs display empform;
 exec frs initialize;
 exec frs activate menuitem 'Browse';
exec frs begin;
    Browse and update the data on the form;
exec frs end;
 exec frs activate menuitem 'Quit';
exec frs begin;
    exec frs breakdisplay;
exec frs end;
 exec frs finalize;
```

Example 2:

The table field is unloaded within the Scan menu section. If an error is detected, **breakdisplay** terminates both the display and the unloadtable loops.

```
exec frs display empform;
 ...
 exec frs activate menuitem 'Scan';
exec frs begin;
    ...
    exec frs unloadtable empform employee
        (:child = child, :age = age);
    exec frs begin;
        ...
        if (error) then
           exec frs message 'Aborting scan on error';
           exec frs sleep 2;
           exec frs breakdisplay;
        end if;
    exec frs end;
exec frs end;
 exec frs finalize;
/* breakdisplay transfers control here. */
```

Example 3:

Use the breakdisplay statement in a display submenu display block.

```
exec frs display 'form';
 exec frs activate menuitem 'Utilities';
exec frs begin;
    exec frs display submenu;
    exec frs activate menuitem 'Delete';
    exec frs begin;
        do delete based on data on form;
    exec frs end;
    exec frs activate menuitem 'File';
    exec frs begin;
        Place data on form into a file;
    exec frs end;
    exec frs activate menuitem 'End';
    exec frs begin;

        /* Exit from the submenu display block */
        exec frs breakdisplay;
    exec frs end;
    exec frs finalize;
exec frs end;
exec frs activate menuitem 'Done';
exec frs begin;
    /* exit from form display block */
    exec frs breakdisplay;
 exec frs end;
 exec frs finalize;
```

clear Statement—Clears the Screen or Fields

This statement clears the screen, individual fields on the form, or all fields on the form.

Syntax

```
clear screen
clear field fieldname {, fieldname}|all
```

Description

The clear statement clears the screen or the fields on a form. You can issue a clear screen statement anywhere in a forms-based program. (To clear a single row, use the clearrow statement.)

You can specify *fieldname* using a quoted or unquoted character string or host string variable. *Fieldname* can specify a simple field or a table field; if you specify a table field, clear field clears the contents of the associated data set as well as the display. The *fieldname* cannot specify a derived field; clear field cannot clear derived fields.

To clear the screen, use the clear screen option. Clear screen does not remove any data from the form's fields; if you redisplay the form, the fields displays the same data they held when the screen was cleared.

The clear field option clears data from the specified *fieldnames* and sets each cleared field's change variable to 0. Clear field all clears all field change variables, but does not clear the form's change variable. Derived fields that depend on a field that has been cleared are also cleared. Unlike the clear screen option, data in the cleared fields is not recoverable.

By default, when you clear a nullable field, blanks are placed in the field's display. You can override this default by defining the logical environment variable, II_NULL_STRING. When you do so, the value contained in II_NULL_STRING is placed in the field instead of blanks.

You must issue the clear field statement within the execution scope of a form's display loop. However it is not required to be syntactically within the display block: the clear field statement can be executed in a routine or procedure that is called from within the display loop.

Examples—clear statement:

Example 1:

Present the user with a blank screen, followed by a message.

```
exec frs clear screen;
exec frs message 'Initializing ...';
```

Example 2:

Clear the data in the salary and dept fields.

```
exec frs clear field salary, dept;
```

Example 3:

Clear the data in the field specified by the program variable fieldvar.

```
exec frs clear field :fieldvar;
```

Example 4:

Clear the data set of the employee table field.

```
exec frs clear field employee;
```

clearrow Statement—Clear Data from a Field

This statement clears data from a row in the table field.

Syntax

```
clearrow formname tablename [row]
    [(columnname {, columnname})]
```

Description

The clearrow statement clears some or all of the column values from a single row in a table field display and sets the change variable of each cleared column to 0. This statement does not delete a row, but simply clears its values.

This statement does not clear derived columns directly. That is, the clearrow statement skips any derived columns when you clear all the columns. Additionally, the statement does not accept a derived column name in its column list. However, when you clear a source column of a derived column, the value in the derived column is then invalid and is cleared also. (For a discussion of derived and source columns, see Table Fields (see page 579).)

By default, after you execute a clearrow statement, the nullable columns contain blanks. You can override this default by defining II_NULL_STRING. When you do so, the value contained in II_NULL_STRING is placed in the nullable columns instead of blanks.

Formname is the name of the form in which the table field *tablename* is displayed. You can use quoted or unquoted character strings or program variables to specify *formname* and *tablename*.

Row is the number of the displayed row to be cleared. It can be either an integer literal or an integer program variable. The *row* must be in the range from 1 to the number of displayed rows for the table field as specified in VIFRED. If it is omitted, the row on which the screen cursor currently rests is cleared.

If you include the *columnname* list, the statement clears only the values in the specified columns for the specified row. *Columnname* can be specified using a character string, with or without quotes, or as a program variable. You cannot specify a derived or hidden column.

If you want to remove all rows in a data set, use the clear statement; use the deleterow statement to delete a row.

Examples—clearrow statement:

Example 1:

Clear all values in the current row.

```
exec frs clearrow empform employee;
```

Example 2:

Clear the ename column in the first displayed row.

```
exec frs clearrow empform employee 1 (ename);
```

deleterow Statement—Delete a Row

This statement deletes a row from the table field.

Syntax

`deleterow` *`formname tablename`* `[`*`row`*`]]`

Description

The deleterow statement deletes a row from the table field display and marks its state as DELETED. The remaining rows in the table field are compacted. The next row in the data set is immediately scrolled into the display.

Deleting a (changed or unchanged) row from a table field data set moves that row to the end of the data set. The values in deleted rows are still accessible to the program (using unloadtable), though the rows no longer appear in the table field display. However, when you delete rows having a state of new or undefined, those rows are removed from the data set, and are not accessible to the application program.

The FRS does not perform an automatic validation check before deleting the row.

The *formname* identifies the form in which the table field *tablename* is displayed. Use character string literals, quoted or unquoted, or program variables to represent either.

The *row* is the number of the displayed row to be deleted. It can be either an integer literal or an integer program variable. It must evaluate to a number not smaller than 1 nor greater than the total number of displayed rows in the table field. If you do not include *row*, the row on which the cursor is currently resting is deleted.

Examples—deleterow statement:

Example 1:

The Delete menu item deletes the current row and saves the employee number, for use in updating the database (unless the row was a NEW row, appended by the runtime user, in which case the row is not used to update the database).

```
exec frs activate menuitem 'Delete';
 exec frs begin;
    exec frs getrow (:eno = eno, :state = _state);
    exec frs deleterow empform employee;

    if (state = 2 or state = 3) then
        /*
        ** The state indicates that the row was loaded
        ** by program from the database. (It may have
        ** since been modified.)
        */
        exec sql delete from employee
            where eno = :eno;
    end if;
exec frs end;
```

Example 2:

Get null values.

```
exec frs getrow (:spouse_var:indicator_var = spouse,
                 :age_var = age)
exec frs deleterow empform employee 1
```

describe Statement—Retrieve Descriptive Information

This statement retrieves descriptive information about a form or table field.

Syntax

For retrieving form information:

```
describe form formname [mode]
    into descriptor_name
```

For retrieving table field information:

```
describe table formname tablename [mode]
    into descriptor_name
```

Description

The forms describe statement is used in dynamic applications to return the data type, length, and name information about the simple fields on a form or the columns of a table field. This information, which is stored in the SQLDA, is used by dynamic forms input and output statements to set and retrieve form data.

You can issue the describe statement at any time after the form is initialized. (The forminit or addform statement initializes a form.) It is not necessary to initialize a table field to describe it, although the form which contains the table field must have been initialized.

Note: The describe statement is not available in QUEL. For dynamic behavior in QUEL use the param statement. See the *QUEL Reference Guide*.

Describe statements can be issued from inside a display loop even if the form described is the form being displayed.

Formname identifies the form being described and can be a quoted or unquoted string or program string variable.

Tablename identifies the table field being described and must be a table field associated with *formname*. *Tablename* can be a string, quoted or unquoted, or a program string variable.

Mode determines what subset of fields or columns is described. You can specify *mode* with a quoted or unquoted string or program string variable. Its value can be any of the following:

update

Specifies that the FRS returns the description of all updatable fields in a form or columns in a table field. Descriptions of derived, display-only, and query-only fields are not returned. This mode corresponds to a form or table field displayed in update or fill mode.

query

Specifies that the FRS returns the description of updatable and query-only fields in a form or columns in a table field. This mode corresponds to a form or table field displayed in query mode.

all

Specifies that the FRS returns a description of all fields in the form or columns in the table field, including those that are display-only or query-only. You can use this mode to map complete database table descriptions (such as those returned by describing the prepared statement 'select * from emp') to a form or table field.

If no *mode* is specified, then the *mode* defaults to all.

Descriptor_name is the name of an SQL Descriptor Area (SQLDA). An SQLDA is a host language structure allocated at run time whose name, if not SQLDA, is defined by the program. Part of the structure of an SQLDA is an array of **s**qlvar elements. After a form or table field is described, each sqlvar element contains the data type, length, and name of one of the fields in the form or columns in the table field corresponding to the specified *mode*. The FRS does not return hidden column names nor the internal row variables _state and _record in a table. If you want to use these variables in an application, the application must allocate a sqlvar element for each individually.

If the returned data type code indicates a table field when describing a form, the program must issue a describe table statement to collect the column information about the table field. You can issue this statement at the end of the processing or immediately with a different SQLDA.

Each host language has different considerations for the SQLDA structure; in some languages it is not defined. See the *SQL Reference Guide* for a complete description of the structure of an SQLDA and its use in a dynamic forms application. Your host language companion guide has information about allocating an SQLDA and the variables associated with it.

Example—describe statement:

This program fragment generates a report from a form. The form has both simple fields and a table field.

```
...
exec frs describe form :form_var into :form_desc;
/* Confirm that form descriptor is large enough */
if (form_desc.sqld > form_desc.sqln) then
    free form_desc;
    allocate a descriptor with SQLD
        or more SQLVAR elements;
    exec frs describe form :form_var into :form_desc;
end if;
/* Generate report for form */
call report_form(form_var, form_desc);

/* Find all table fields (type 52), describe them
** and generate a report */
for index = 1 to form_desc.sqld loop
    if (form_desc.sqlvar(index).sqltype = 52) then
        table_var = form_desc.sqlvar(index).sqlname;
        exec frs describe table :form_var :table_var
                    into :table_desc;

        /* Confirm that table field descriptor
        ** is large enough */
            if (table_desc.sqld > table_desc.sqln)
                then
                free table_desc;
                allocate a descriptor with SQLD
                     or more SQLVAR elements;
                exec frs describe table :form_var
                    :table_var into :table_desc;
            end if;
        call report_table(table_var, table_desc);
    end if;
end loop;
...
```

display Statement—Initiate Form Display

This statement initiates the display of a form.

Syntax

```
display formname [mode]
    [with style=fullscreen[(screenwidth = value)]
    | popup [(option=value {, option=value})]]
```

Description

The display statement marks the beginning of the form's display block in the program and, when executed, initiates the form's display loop. (For a description of display blocks and loops, see Embedded Forms Program Structure (see page 541).)

The form, identified by *formname*, appears on the screen in the specified *mode*. Four modes are available:

fill

Displays data on the screen and allows the data to be modified by the user. Data displayed when the form first appears comes from one of two sources:

- The default values specified in the form's VIFRED definition
- Values placed in fields by the initialize statement

The fill mode clears all fields on the form except those containing VIFRED-defined default values before any initialization code is executed. This is the default mode.

update

Like fill mode, displays data that the user can modify.

The update mode does not clear the form's fields before executing the initialization code. Consequently, when you display a form in update mode, the fields retain any values entered into them by data transfer statements (such as a putform) or present when a previous display terminated. Even clearing a field on the screen does not clear the data that is redisplayed when the form is called again.

read

Lets the user examine but not change the data on the form

query

Presents a form with cleared fields so that users can enter data to construct a query on the database. This mode allows users to enter comparison operators (for example, "<>" or ">" in addition to normal data. (The creator of the form must supply the extra space in a field for the operators.)

If you display a form in fill, update, or read mode, any derived fields on the form are calculated and displayed, if possible. (For more information about derived fields, see Table Fields (see page 579).) In query mode, derived fields are displayed with blanks.

If no mode is specified in the display statement, the mode defaults to fill.

You can express both *formname* and *mode* as character strings, with or without quotes, or as program string variables.

Including the style clause allows you to override the form's default style definition as specified in VIFRED. The form displays in either fullscreen or pop-up, as specified, regardless of the style specified with VIFRED.

If you choose the fullscreen style, the form is displayed over the entire screen. However, the optional screenwidth argument lets you choose the effective width of your terminal screen. Your fullscreen form can appear on a narrow or wide screen. The actual width of each choice differs depending on your terminal.

The *value* assigned to screenwidth can be an integer or string literal or an integer variable. It must be one of the following values:

default | 0

Specifies that the FRS uses the screenwidth option that was defined in VIFRED

current | 1

Specifies that the screenwidth remains as it is

narrow | 2

Specifies the smaller of two possible widths

wide | 3

Specifies the wider of two possible widths

A pop-up form covers only part of the screen. It is most useful when you want to display a form on top of another form. Also, because they do not cover the entire screen, you can display more than one pop-up at a time. The only limit on the number of pop-up forms that can be active at one time is the amount of available memory.

The *option* arguments specify the screen location and border style of a pop-up form. The *options* are:

Startcolumn

Specifies the column position of the upper left hand corner of the pop-up form

Startrow

Specifies the row position of the upper left hand corner of the pop-up form

Border

Determines the type of border around the form

The value assigned to the startcolumn or startrow option can be an integer literal, an integer variable, or either of the keywords, default or floating. Values assigned as integer literals or by means of variables can be any of the following:

>0

Specifies that values greater than 0 are interpreted as terminal screen coordinates. (The terminal screen origin, in the upper left hand corner, is defined as 1,1. Values increase to the right and down the screen.)

0

Specifies that a value of 0 is equivalent to the keyword default. The FRS checks to see if the option has a value defined in VIFRED. If it does, then the FRS uses that value. If not, FRS selects a value near the current field in the parent form.

-1

Specifies that the value -1 is equivalent to the keyword floating. The FRS chooses a value close to the current field in the parent form.

If the values you select for the options do not allow the entire form to be displayed on the screen, the FRS tries to adjust the starting location to do so. If the form is larger than the screen, the form is displayed in fullscreen style.

You can assign the border option an integer literal value, an integer variable, or one of the keywords, default, none, or single. Default uses the form's default border definition. If the form does not have a default border definition, the default is a simple, single-line border. None eliminates the pop-up's border. Single forces the use of a single-line border, regardless of the default border definition.

The integer literal or variable can be one of three values, each equivalent to one of the keywords:

0

Is equivalent to the keyword default

1

Is equivalent to the keyword none

2

Is equivalent to the keyword single

If you do not specify the *options* when you display a form with style = popup, the FRS uses the pop-up defaults rather than any corresponding defaults specified for the form when it was created. The pop-up defaults are a starting location as close as possible to the current field (defined as the field on which the cursor currently rests) and a single-line border.

If the style clause is not specified, the form's default style (specified with VIFRED) is used to determine the display style.

Examples—display statement:

Example 1:

Display empform in update mode, with only an initialize and finalize statement.

```
exec frs display empform update;
 exec frs initialize (ename = :ename, age = :age);
exec frs finalize (:ename = ename, :age = age);
```

Example 2:

Display empform in the default fill mode, with two activate sections and no initialize or finalize statements.

```
exec frs display empform;
 exec frs activate menuitem 'Fill';
exec frs begin;
    program code;
exec frs end;
 exec frs activate menuitem 'End';
exec frs begin;
    exec frs breakdisplay;
exec frs end;
 program code;
```

Example 3:

Display empform in a pop-up window, using the default border and placing the upper left hand corner of the form in the fifth column and fifth row.

```
exec frs display empform with style=popup
    (startcolumn=5, startrow=5, border=default);
```

Example 4:

Display empform, setting the terminal screen to narrow width.

```
exec frs display empform with
       style=fullscreen(screenwidth=narrow);
```

display submenu Statement—Display a Submenu

This statement initiates the display of a submenu.

Syntax

```
display submenu
```

Description

The display submenu statement provides a way to nest actions for a form. When a display submenu is started, the current set of actions is suspended. The suspended actions are reinstated after the display submenu exits.

You can nest a display submenu loop inside another display submenu loop, but the outermost loop must be inside a display loop. A display submenu loop looks and behaves exactly like a display loop. This means that activate field statements are allowed and the resume menu statement works just as it does in a display loop. For more information about display loops and display submenu loops, see Embedded Forms Program Structure (see page 541).

(If you want to put a menu on the screen that is not associated with any form, use the submenu statement. For information about submenus and how they differ from display submenus, see Embedded Forms Program Structure (see page 541). See also submenu Statement—Start a Menu Not Linked to a Form (see page 745).)

Example—display submenu statement:

This example demonstrates using the display submenu statement to provide a submenu associated with the menu item fld1.

```
...
 exec frs activate menuitem 'fld1';
 exec frs begin;
 /* Start of a display submenu loop */
exec frs display submenu;
exec frs activate menuitem 'fldx';
exec frs begin;
     process field x;
exec frs end;
exec frs activate menuitem 'fldy';
exec frs begin;
     process field y;

     /* Exit display submenu with no validation */
     exec frs breakdisplay;
exec frs end;
exec frs activate menuitem 'end';
exec frs begin;
     /* Exit display submenu with validation */
     exec frs enddisplay;
 exec frs end;
     /* End of activation block for menuitem 'fld1' */
 exec frs end;
```

enddisplay Statement—Terminate a Display

This statement terminates a display or display submenu loop, performing field validation and executing the finalize statement.

Syntax

```
enddisplay
```

Description

The enddisplay statement terminates a display loop or a display submenu loop, returning control to the first statement in the program following the end of the display or display submenu block.

When the FRS executes the enddisplay statement, it performs a validation check on the data in all the fields in the form, using the validation criteria specified when the form was defined in VIFRED. If any invalid data is found, the enddisplay statement is aborted and the display loop resumes with the cursor positioned on the first field containing invalid data. Once the data has been corrected, the operation containing the enddisplay statement can be retried by the user.

If the data validation is successful, enddisplay executes the finalize statement if one is present in the display block and then terminates the display loop.

The enddisplay statement must be syntactically within the scope of a display block, as it generates a local goto statement to the end of the block. If the enddisplay statement is in the program loop of an unloadtable statement nested within an activate section, enddisplay exits the unloadtable loop, in addition to ending the display loop. (The second example demonstrates this.)

To terminate the display loop without validating data or executing the finalize statement, use the breakdisplay statement.

Examples—enddisplay statement:

Example 1:

The End operation terminates display of the form.

```
exec frs display empform;
exec frs initialize;
exec frs activate menuitem 'Browse';
exec frs begin;
    Browse and update the data on the form;
exec frs end;
exec frs activate menuitem 'End';
exec frs begin;
    exec frs enddisplay;
exec frs end;
exec frs finalize (:ename = ename, :age = age);
```

Example 2:

The table field is unloaded within the Scan menu section. If a certain condition is detected, enddisplay ends both the display and the unloadtable loops.

```
exec frs display empform;
    ...
exec frs activate menuitem 'Scan';
exec frs begin;
      ...
     exec frs unloadtable empform employee
         (:child = child, :age = age);
     exec frs begin;
         ...
         if (condition is true) then
             exec frs message 'Ending the scan';
             exec frs sleep 2;
             exec frs enddisplay;
         end if;
    exec frs end;
exec frs end;
exec frs finalize (:ename = ename, :dept = dept,
                  :sal = sal);
/* enddisplay transfers control to the
** finalize statement above.
*/
```

Example 3:

Use the enddisplay statement in a display submenu display block.

```
exec frs display 'form';
 exec frs activate menuitem 'Utilities';
exec frs begin;
    exec frs display submenu;
    exec frs activate menuitem 'Delete';
    exec frs begin;
        do delete based on data on form;
    exec frs end;
    exec frs activate menuitem 'File';
    exec frs begin;
        place data on form into a file;
    exec frs end;
    exec frs activate menuitem 'End';
    exec frs begin;
```

```
            /* exit from the submenu display block */
            /* after validating data on form */
            exec frs enddisplay;
        exec frs end;
        exec frs finalize (:balance_var = balance);
exec frs end;
exec frs activate menuitem 'Done';
exec frs begin;
    /* exit from form display block */
    /* after validating data on form */
    exec frs enddisplay;
exec frs end;
 exec frs finalize (:total_var = total);
```

endforms Statement—Terminate Application Connection

This statement terminates the application's connection with the FRS.

Syntax

```
endforms
```

Description

The endforms statement terminates an application's connection with the FRS. Endforms does not clear the terminal screen.

Endforms must be the last forms statement executed in an application. Generally, you must issue the endforms statement when the application is going to terminate. Do not issue the forms and endforms statements more than once in an application.

The forms and endforms statements do not constitute a syntactic block; these statements establish a runtime scope within which you can issue forms statements.

Examples—endforms statement:

Example 1:

Sever the application's connection with the FRS.

```
exec frs forms;
    /*
    ** Forms statements are allowed only
    ** within this block.
    */
exec frs endforms;
```

Example 2:

Clear the screen before exiting from the FRS and the database.

```
exec frs clear screen;
exec frs message 'Exiting application';
exec frs endforms;
exec sql disconnect;
```

endloop Statement—Terminate a Loop

This statement terminates a loop.

Syntax

```
endloop
```

Description

The forms endloop statement terminates the loops defined by the begin/end program blocks associated with several of the forms statements, such as unloadtable, formdata, or tabledata.

If loops are nested, endloop terminates only the loop in which it is executed; no outer loops are terminated. Control is returned to the first statement following the terminated loop. For this reason, you must place the endloop statement syntactically within the loop that it is intended to end.

You cannot use endloop to explicitly terminate a display loop. However, if a display loop is nested within a loop defined by a begin/end block, terminating the loop defined by the begin/end block terminates the nested display loop as well.

When you use endloop to exit a program block loop, the FRS does not validate any data.

Examples—endloop statement:

Example 1:

Break out of the unloadtable loop on an error.

```
exec frs unloadtable empform employee
    (:ename = ename);
exec frs begin;
    program code;
    if (error) then /* break out of UNLOADTABLE */
        exec frs endloop;
    end if;
exec frs end;
```

Example 2:

This example nests a display loop within an unloadtable statement. The endloop statement implicitly breaks out of the nested display loop. The example assumes a form formnames with a table field formtable that the user fills with different form names and modes.

```
exec frs unloadtable formnames formtable
    (:fname = fname, :fmode = fmode);
exec frs begin;
    exec frs display :fname :fmode;
    exec frs initialize;
    exec frs activate menuitem 'Next';
    exec frs begin;
        exec frs breakdisplay;
    exec frs end;
    exec frs activate menuitem 'Quit';
    exec frs begin;
        exec frs endloop;
    exec frs end;

 exec frs finalize;
 /* breakdisplay transfers control to here */
    exec frs message 'Next form';
    exec frs sleep 2;
/* endloop transfers control to here */
exec frs end;
```

finalize Statement—Transfer Data

This statement performs final data transfers at end of display loop.

Syntax

```
finalize [(variable[:indicator_var] = fieldname
    {, variable[:indicator_var] = fieldname})]
```

Description

The finalize statement transfers data from simple fields on a form into program variables at the end of the display loop. (You can transfer data from table field columns by means of the getrow and unloadtable statements.) Data is transferred from each specified field, identified by *fieldname*, into its associated variable.

You can specify *fieldname using* a character string, with or without quotes, or as a program string variable. The *variable* associated with each field must be a data type compatible with the field.

You must include the indicator variable if the field is nullable. If the retrieved data is a null, the indicator variable is set to -1. (If the retrieved data is a null and an indicator variable is not present, Ingres returns an error.)

Using an indicator variable can also provide a useful program check when the field holds character data even if the field is not nullable. When the retrieved character string is larger than the variable to which it is assigned, the string is truncated. In such instances, if an indicator variable is present, it is set to an integer indicating the full, untruncated length of the character string.

The finalize statement (if present) is executed when a form's display is ended by the enddisplay statement. If the display ends due to breakdisplay statement or if enddisplay is not executed for any reason, the finalize statement is not executed.

The FRS also executes the finalize statement when the user selects the Menu key to end the display. (The Menu key is used to end the display of a form that has no defined menu item operations and, thus, no menu items.)

The finalize statement is not required. However, even when it does not perform any data transfers, it provides a useful way to mark the end of the display block.

Examples—finalize statement:

Example 1:

Place data from the name and salary fields into program variables.

```
exec frs finalize (:namevar = name,
    :salvar = salary);
```

Example 2:

Place data from the field specified by the variable fieldvar into the variable namevar.

```
exec frs finalize (:namevar = :fieldvar);
```

Example 3:

Get values from a nullable field.

```
exec frs finalize (:spousevar:indicator_var = spouse);
```

formdata Statement—Loop Through Fields

This statement loops through all fields in the form.

Syntax

```
formdata formname
begin
    program code;
end
```

Description

The formdata statement loops through the fields on a specified form, executing the code in the begin/end block for each field. *Formname* identifies the form and can be expressed as a character string, with or without quotes, or as a program string variable.

This statement is used primarily with the inquire_frs statement to allow inquiries to be easily made on all fields in a form. This is particularly useful if the program does not know until runtime what forms it uses. Formdata causes the same section of code to be executed for all the fields on the form; therefore, the notion of the *current* field allows inquire_frs to be effectively used.

The template for this usage is:

```
formdata formname
begin
    inquire_frs on current object;
    program code
end
```

The syntax of the inquire_frs statement is described in inquire_frs Statement—Retrieve FRS Runtime Information (see page 673).

The full range of inquire_frs statements, as well as all other embedded SQL and host language statements, can appear within the formdata loop. The formdata loop starts the first pass on the first field in the form and sequences along to the next field, including display-only fields, with each additional pass through the loop.

You can use the formdata statement in conjunction with tabledata to loop through all columns in a table field. See the example below and the tabledata statement description for details.

You cannot issue a formdata statement for a form inside that form's display loop, nor can you initiate a display loop for a form inside a formdata loop of the same form.

To terminate the formdata's program loop, use the endloop statement.

Example—formdata statement:

Loop through a form, printing out all field and column names. The blank strings in the inquire_frs reference the current field and the current table field column, respectively.

```
exec frs formdata :formname;
exec frs begin;
    exec frs inquire_frs field ''
        (:fldname = name, :istable = table);
    if (istable = 1) then
        print fldname,' is a table field';
        print '---------------';
        exec frs tabledata;
        exec frs begin;
            exec frs inquire_frs column '' ''
                (:colname = name);
            print colname, ' is a column';
        exec frs end;
        print '---------------';
    else
        print fldname, ' is a regular field';
    end if;
exec frs end;
```

forminit Statement—Declare a Form

This statement declares a form to the FRS.

Syntax

```
forminit formname {, formname}
```

Description

The forminit statement declares a form to the FRS. Declaring a form makes its definition known to the FRS. You must declare a form before you can use it in an application. However, after a form has been declared, the application can display it any number of times. Forminit is used only to declare uncompiled forms. If you are declaring a form that was compiled in VIFRED, you must use the addform statement.

Because the forminit statement retrieves the form's definition from the forms catalogs in the database, you must issue it after the application connects to the database. Do not place this statement in a host variable declaration section.

Formname identifies the form and can be specified using a character string, with or without quotes, or as a program string variable.

If no transaction is open when your application issues the forminit statement, then forminit issues a commit afterwards, which releases all its shared locks. If a transaction is open, then forminit does not commit afterwards and your application continues to hold shared locks on the forms system catalogs until the next commit or end transaction statement.

Forminit takes shared locks on the forms system catalogs. These locks allow others to access the forms in the catalogs but not to change them. If the form has a VIFRED table look-up validation defined for it of the type field in table.column, then forminit also takes shared locks on the table specified in the validation.

Because of this, applications must, whenever possible, commit open transactions prior to issuing a forminit statement.

Examples—forminit statement:

Example 1:

Declare the forms empform and deptform.

```
exec frs forminit empform, deptform;
```

Example 2:

Inside a local procedure, which is called more than once, control the reissuing of the forminit statement. Drop shared locks on forms system catalogs by issuing commit after the forminit.

```
added integer; /* statically initialized to 0 */
...
if (added = 0) then
    exec frs message 'Initializing form...';
    exec frs forminit empform;
    exec sql commit;
    added = 1;
end if;
```

forms Statement—Invoke the FRS

This statement invokes the FRS for the application.

Syntax

```
forms
```

Description

The forms statement invokes the FRS for a forms-based application. The forms statement must be the first forms statement issued in a forms-based application. Do not issue the forms and endforms statements more than once in an application.

The forms statement clears the screen.

The forms and endforms statements do not constitute a syntactic block, but do set up a runtime scope in which forms statements can be issued.

Restrictions

The forms statement has the following restrictions:

- Programs that invoke the FRS cannot be run in batch.
- When connected to the FRS, program must not issue any non-FRS terminal I/O statements (for example, curses functions in an embedded C program).

Examples—forms statement:

Example 1:

Invoke the FRS for a forms application.

```
exec frs forms;
 /*
** Forms statements are allowed only within this
** block.
*/
exec frs endforms;
```

Example 2:

Start an application.

```
exec frs forms;
 exec frs message 'Initializing database and forms
...';
exec sql connect personnel;
exec frs forminit empform;
exec sql commit;
```

getform Statement—Transfer Data into Program Variables

This statement transfers data from the form into program variables.

Syntax

Non-dynamic version:

```
getform [formname]
     (variable[:indicator_var] = fieldname
     {, variable[:indicator_var] = fieldname})
```

Dynamic version:

```
getform [formname]
     using [descriptor] descriptor_name
```

Description

The getform statement transfers data from simple fields on the specified form into program variables. (You can transfer data from table field columns using the getrow and unloadtable statements.) *Formname* must specify a declared form (see the addform and forminit statements) that has data in its fields. You can specify *formname* using a quoted or unquoted character string or host string variable.

Getform does not return operators entered into forms in query mode; for example, if the user enters >1000, getform returns 1000. To obtain the operator (>), you must use the getoper statement.

If you do not specify a *formname*, getform must be syntactically within a display block, and the current form is assumed.

Fieldname must identify a field on the specified form. You can specify *fieldname* using a quoted or unquoted character string literal or as a program variable. The variable associated with the field must have a data type compatible with the field's data type. (See your host language companion guide for information about compatible data types.)

You must include the *indicator variable* if the field is nullable. If the retrieved data is null, the indicator variable is set to -1. (If the retrieved data is null and an indicator variable is not present, Ingres returns an error.)

Using an indicator variable also enables your application to detect string truncation. When the retrieved character string is larger than the variable to which it is assigned, the string is truncated. In such instances, if an indicator variable is present, it is set to an integer indicating the full, untruncated length of the character string.

The dynamic version of the getform statement transfers data from the specified form to variables pointed at and described by *descriptor_name*. *Descriptor_name* identifies an SQLDA (SQL Descriptor Area), a host language structure allocated at run time. (The actual structure name is not required to be SQLDA and can be differently defined by the program.)

You must describe the form and allocate the variables pointed at by the SQLDA before you can issue a dynamic getform statement. Read the *SQL Reference Guide* and your host language companion guide for information about the structure, allocation, and use of the SQLDA.

Note: The dynamic version of the getform statement is unavailable in QUEL. For dynamic behavior in QUEL use the param statement. See the *QUEL Reference Guide*.

Getform validates the fields before retrieving the values. If a field contains invalid data, a runtime error is displayed and the user variable is not updated. However, execution flow is unaffected.

Examples—getform statement:

Example 1:

Place data from the ename and sal fields of form empform into program variables.

```
exec frs getform empform (:namevar = ename,
                              :salvar = sal);
```

Example 2:

Place data from the field specified by the variable fieldvar in the form specified by the variable formvar, into the variable namevar.

```
exec frs getform :formvar (:namevar = :fieldvar);
```

Example 3:

Place data from the current form into a database table. Within a display block, the form name need not be specified.

```
...
 exec frs activate menuitem 'Add';
exec frs begin;
    exec frs validate;
    exec frs getform (:namevar = ename,
                      :salvar = sal);
    exec sql insert into employee (ename, sal)
        values (:namevar, :salvar);
exec frs end;
```

Example 4:

Get values from a nullable field.

```
exec frs getform empform (:spousevar:indicator_var
                                           = spouse);
```

Example 5:

Using a dynamic getform statement, retrieve a query operator and a salary value from the sal field, displayed in query mode.

```
sqlda.sqld = 2;
 sqlda.sqlvar(1).sqltype = INT;
sqlda.sqlvar(1).sqllen = 4;
sqlda.sqlvar(1).sqldata = address(op_var);
 sqlda.sqlvar(1).sqlind = null;
sqlda.sqlvar(1).sqlname = 'GOP(sal)';
 sqlda.sqlvar(2).sqltype = -FLOAT;
sqlda.sqlvar(2).sqllen = 8;
sqlda.sqlvar(2).sqldata = address(sal_var);
sqlda.sqlvar(2).sqlind = address(null_ind);
slqda.sqlvar(2).sqlname = 'sal';
 exec frs getform :form_var using descriptor sqlda;
```

getoper (function) Statement—Get Comparison Operators

This statement gets comparison operators from fields and columns displayed in query mode.

Syntax

```
getform formname (oper_variable = getoper(fieldname))

getrow formname tablename [row]
    (oper_variable = getoper(columnname))
```

Description

The getoper function returns an integer code indicating which comparison operator was entered into a form field or table field column. When a form or table field is in query mode, users can enter comparison operators, such as > or <, into the form's fields or table field columns. In addition, the user can enter data into fields. The program uses the operators and data entered by the user to build a query. While the actual data can be retrieved for the query with any statement that retrieves data from a form field or table field row, you must use the getoper function, in conjunction with these statements, to retrieve comparison operators.

The getoper function can appear within the context of all the forms statements that retrieve data from a form field or a table field column. Those statements are:

- **getform**
- **finalize**
- **getrow**
- **unloadtable**

For a discussion of the syntax of each of these, see their individual statement descriptions. The syntax does not differ when the getoper function is used with these statements.

There are three problems in constructing a query based on what the user has entered on a form:

- The program must find out whether a user has entered any data in the fields or columns. Any fields that the user left empty must be ignored in constructing retrieval qualifications (where clauses).
- The program must be able to find what operator, if any, the user typed alongside the data in a particular field or column.
- Given the data in the field or column, the program must construct a where clause to qualify a database retrieval.

The getoper function provides the solution to the second problem. This function returns to the integer variable *oper_variable* a query operator code that represents the operator it finds on the specified field or column. The following are the codes that getoper returns:

Operator Code	Operator Entered in Field or Column
0	No data entered in field or column
1	=
2	<> or !=
3	<
4	>
5	<=
6	>=

If a user types data without an explicit operator, this implies an operator of =, and getoper returns the value 1. If the form is not displayed in query mode, the value 0 is returned.

The getoper function can appear in a forms statement together with the retrieval of other objects. In fact, in most cases, a program retrieves both the operator and value of a particular field or column in a single statement. For example,

```
exec frs getform empform
      (name_oper = getoper(ename), vname = ename,
       age_oper = getoper(age), vage = age);
```

The information returned by getoper, together with the other values retrieved from the form, can be used by a program to construct a character string that represents a qualification.

Getoper causes the FRS to perform data type checking on the field. If the data in the field is not the correct data type, for example, if the user has entered character data in a numeric field, then a runtime error is issued and the user variable is not updated. However, program flow is not affected by data type checking errors.

When you initialize a table field in query mode, the FRS automatically takes care of the bookkeeping chores when the table field scrolls. For example, if a number of rows are displayed, and the user scrolls in a new row, the first row with its data *and* its comparison operators are scrolled up. Consequently, when the getoper function is used with the unloadtable statement, the columns and operators being retrieved need not be currently displayed. A general rule to follow when constructing queries from the getoper function used within an unloadtable statement is to qualify columns and or rows.

For example, the following data set constructs the specified qualification:

	eno	**sal**
row 1	<=20	
row 2		>50000
row 3	>=40	<35000

Qualification:

```
where (e.eno <= 20) or
      (e.sal > 50000) or
      (e.eno >= 40 and e.sal < 35000)
```

If a table field is not in query mode, then the value 0 is returned and a runtime error is issued.

When the getoper function is used in conjunction with a Dynamic FRS using clause and its associated SQLDA, the function name is abbreviated to gop.

Example—getoper (*function*) statement:

Example 1:

Build a query using the getoper function. This example uses a query operator array that can map the integer codes of FRS query operators to their corresponding query language operators. Array subscripts begin at 1.

```
exec sql begin declare section;
    oper_chars     array(6) of character_string(2);
    oper           integer;
    eno            integer;
    enum           character_string(10);
    where_clause   character_string(30);
exec sql end declare section;
 oper_chars = ('=', '!=', '<', '>', '<=', '>=');
 exec frs display empform query; /* Use query mode */
exec frs activate menuitem 'Retrieve';

exec frs begin;
    exec frs getform empform (:eno = eno,
                             :oper = getoper(eno));
    /*
    ** assign value in eno to enum and convert
    ** type to character
    */
    if (oper > 0) then
    /* Construct where clause like:
    ** employee.eno = 18
    */
    where_clause = 'employee.eno '
                    + oper_chars(oper) + enum;

    else
        /* Nothing entered, use 'truth' default */
        where_clause = '1 = 1';
    end if;
    exec sql select eno, ename, age
        into :eno, :ename, :age
        from employee
        where :where_clause;
    exec sql begin;
        process rows;
     exec sql end;
 exec frs end;
```

getrow Statement—Get Values from a Row

This statement gets values from a table field row.

Syntax

Non-dynamic version:

```
getrow formname tablename [row]
    (variable[:indicator_var] = columnname
    {, variable[:indicator_var] = columnname})
```

Dynamic version:

```
getrow formname tablename [row]
    using [descriptor] descriptor_name
```

Description

The getrow statement transfers values from columns in a table field row into program variables. It does not change any of the values in the row.

The *formname* is the name of the form in which the table field *tablename* is displayed. Both can be expressed as character strings, with or without quotes, or as program variables.

The *row* is the number of the displayed row from which the data is to be transferred. *Row* can be either an integer literal or an integer program variable. *Row* cannot be less than 1 nor greater than the number of currently displayed rows. For example, if the table field can display 7 rows of data, but currently only shows 5, then *row* can have a value ranging from 1 to 5, inclusive. If it is omitted, the values are taken from the row on which the screen cursor currently rests.

The getrow statement transfers values from the specified *columnnames* of the table field row into program *variables*. Each *columnname* can be expressed as a character string, with or without quotes, or as a program variable. The *variable* must be of a type appropriate to the column.

You must include the *indicator variable* if the column is nullable. If the retrieved data is null, the indicator variable is set to -1. (If the retrieved data is null and an indicator variable is not present, Ingres returns an error.)

Indicator variables also enable your application to detect string truncation. When the retrieved character string is larger than the variable to which it is assigned, the string is truncated. In such instances, if an indicator variable is present, it is set to an integer indicating the full, untruncated length of the character string.

A column can be a displayed, invisible, derived, or a hidden column. If the table field is associated with a data set, the column can also be one of the special constants _record or _state. The _record constant returns an integer representing the row's record number in the data set, with 1 signifying the first row in the data set. The data set record number is independent of the row's position in the table field display. The _state constant returns an integer corresponding to the row's state (NEW, UNDEFINED, CHANGED, and so forth). The state indicates the most recent event that has occurred to a row's displayed columns. For instance, a row that was originally loaded by the program and has since been updated by the runtime user has a state of CHANGED, signified by the value 3. Programs often use a row's state when updating a database table from the values in the table field. For a complete description of row states, see Table Fields (see page 579).

The dynamic version of the getrow statement transfers values from columns in a table field row to variables pointed at and described by *descriptor_name*. *Descriptor_name* identifies an SQL Descriptor Area. The SQLDA is a host language structure allocated at run time. The actual structure name is not required to be SQLDA and can be differently defined by the program.

Your program must describe the table field and allocate the variables pointed at by the SQLDA before it can issue a dynamic getrow statement. Read the *SQL Reference Guide* and your host language companion guide for information about the structure, allocation, and use of the SQLDA.

The getrow statement causes the FRS to validate the column before retrieving the values in the column. If the validation fails on any column, a runtime error results and the associated user variable is not updated. However, program flow is not affected.

Getrow does not return operators entered into forms in query mode; for example, if the user entered >1000, getrow returns 1000. To obtain the operator (>) you must use the getoper statement.

(The dynamic version of the getrow statement is not available in QUEL.)

Examples—getrow statement:

Example 1:

Get information from the first row of the table field display.

```
exec frs activate menuitem 'GetFirst';
 exec frs begin;
    exec frs getrow empform employee 1
         :eno = eno, :ename = ename, :age = age,
         :sal = sal, :dept = dept);
exec frs end;
```

Example 2:

Find out if the current row has been modified.

```
exec frs activate menuitem 'RowChanged?';
 exec frs begin;
    exec frs getrow empform employee
        (:ename = ename, :state = _state);
    if (state = 1 or state = 3) then
    /* New or changed */
        msgbuf = 'You have modified/added employee '
                                         + ename;
        exec frs message :msgbuf;
        exec frs sleep 2;
    end if;
 exec frs end;
```

Example 3:

Get values from a nullable column in a table field.

```
exec frs activate menuitem 'MoreInfo';
 exec frs begin;
    exec frs getrow empform employee
        (:ename = ename, :age = age,
         :spouse:indicator_var = spouse);
        /* -1 means no spouse or children */
        if (indicator_var <> -1) then
            find information about children,
                if applicable;
        end if;
    Display more detailed information on retrieved
                                          employee;
exec frs end;
```

Example 4:

Using dynamic statements, describe a table field, retrieve a row together with the _state variable, and if it is an original row, delete the row from the database and the table field.

```
exec frs describe table :form_var :table_var
    into sqlda;
...
 exec frs activate menuitem 'Delete';
exec frs begin;
    /* Add an SQLVAR for _state retrieval */
    sqlda.sqld = sqlda.sqld + 1;
    state_col = sqlda.sqld;
    sqlda.sqlvar(state_col).sqltype = INT;
    sqlda.sqlvar(state_col).sqllen = 4;
    sqlda.sqlvar(state_col).sqldata =
                              address(state_var);
    sqlda.sqlvar(state_col).slqind = null;
    sqlda.sqlvar(state_col).sqlname = '_STATE';
    exec frs getrow :form_var :table_var using
                      descriptor sqlda;

    if (state_var = 3) then
    /* Deleting an original row */
       /* Use the SQLDA (minus the _state variable)
       ** for Dynamic SQL
       */
       sqlda.sqld = sqlda.sqld - 1;
       exec sql execute delete_stmt using
           descriptor sqlda;
    end if;
    /* Delete the row from the table field */
    exec frs deleterow :form_var :table_var;
 exec frs end;
```

helpfile Statement—Display Help

This statement displays a file as help text.

Syntax

```
helpfile subjectname filename
```

Description

The helpfile statement enables you to include a help facility in an application. Executing the helpfile statement causes the specified file, *filename*, to be displayed as help text.

Filename must include the correct specification for the file. If the application is to be run by different users include the full directory specification. If the specified file is not found at run time, a message indicating is displayed that no help is available for the *subjectname*. You can specify *filename* and *subjectname* using quoted string constants or host string variables.

This statement allows the user to display the information placed in the specified file (typically information about the current form) and to the function/control/arrow key mappings (by selecting the Keys menu item). However, the user does not have access to help on field validations. To allow the user access to all aspects of the help facility, use the help_frs statement (see page 667) instead.

To edit the contents of the help file from within an Ingres Forms application, define the logical/environment variable II_HELP_EDIT. If II_EDIT is defined, an extra menu item, Edit, is displayed when you select Help. Selecting the Edit menu item invokes the default system text editor on the current frame's help file. (This feature is useful during application development, and is typically not enabled when running an application in a production setting.)

To change the default text editor, set ING_EDIT to the name of the desired editor.

The contents of a help text file are as follows:

- Help text
- **Subtopic statements** (optional)

Subtopic statements must be specified in the following format:

```
/# subtopic 'listitem' 'filename'
```

where *listitem* is the subtopic you want to be displayed and *filename* is the file containing the corresponding help text.

If you include one or more subtopic statements in a help file, a SubTopics menu item is displayed when the user selects Help. If the user selects SubTopics, a list of subtopics is displayed; the list consists of all the listitems from the subtopics statements in the help file. The user can select a subtopic from the list; the help text from the corresponding file is displayed. The /# subtopic command truncates text lines that are longer than 78 characters, without a warning.

- **Comments** (optional)

Comments in the help text file must be specified as follows:

```
/#-- Comment text goes here
/#-- Each line must be flagged
```

You must precede each line of comments with /#--. Comment lines are not displayed when the user selects Help.

Example—helpfile statement:

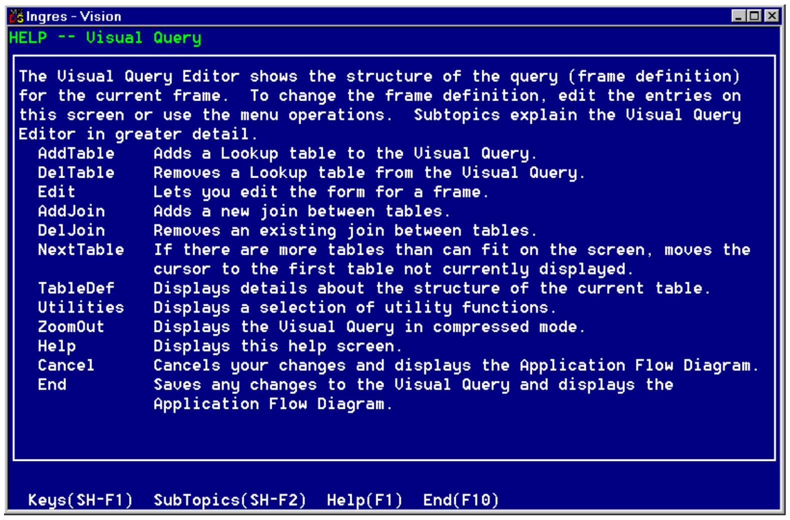
Example 1:

The following help file shows the correct use of the subtopics feature:

```
/#-- vqmain.hlp, the main VQ Editor help screen.
The Visual Query Editor shows the structure of the query (frame definition) for t
he current frame. To change the frame definition, edit the entries on this screen
 or use the menu operations. Subtopics explain the Visual Query Editor in greater
 detail.

AddTable  Adds a Lookup table to the Visual Query.
DelTable  Removes a Lookup table from the Visual Query.
Edit      Lets you edit the form for a frame.
AddJoin   Adds a new join between tables.
DelJoin   Removes an existing join between tables.
NextTable If there are more tables than can fit on the
          screen, moves the cursor to the first table
          not currently displayed.

TableDef  Displays details about the structure of the
          current table.
Utilities Displays a selection of utility functions.
ZoomOut   Displays the Visual Query in compressed mode.
Help      Displays this help screen.
Cancel    Cancels your changes and displays the
          Application Flow Diagram.
End       Saves any changes to the Visual Query and
          displays the Application Flow Diagram.
```

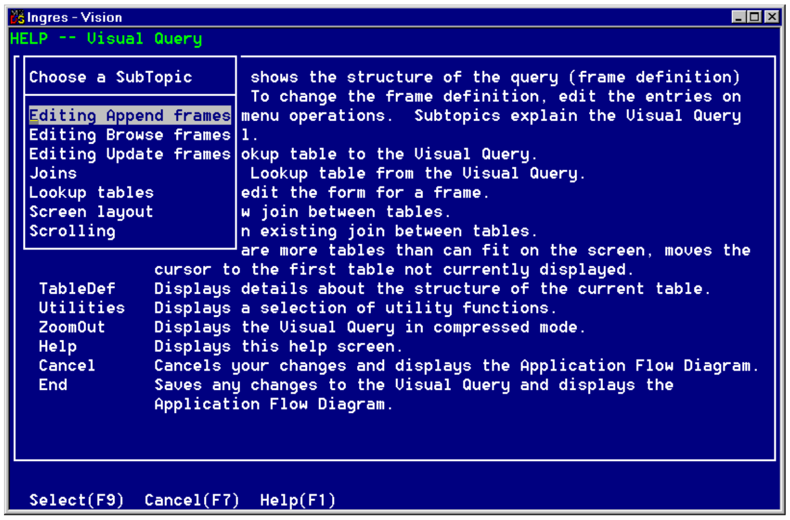
```
/#subtopic 'Editing Append frames' 'vqappend.hlp'
/#subtopic 'Editing Browse frames' 'vqbrows.hlp'
/#subtopic 'Joins' 'vqjoins.hlp'
/#subtopic 'Lookup tables' 'vqlook.hlp'
/#subtopic 'Screen layout' 'vqlayout.hlp'
/#subtopic 'Scrolling' 'vqscroll.hlp'
/#subtopic 'Editing Update frames' 'vqupdat.hlp'
```

When you press Help on the screen that uses the preceding help file, the help text is displayed as shown in the following figure:



```
Ingres - Vision
HELP -- Visual Query

The Visual Query Editor shows the structure of the query (frame definition)
for the current frame.  To change the frame definition, edit the entries on
this screen or use the menu operations.  Subtopics explain the Visual Query
Editor in greater detail.
  AddTable     Adds a Lookup table to the Visual Query.
  DelTable     Removes a Lookup table from the Visual Query.
  Edit         Lets you edit the form for a frame.
  AddJoin      Adds a new join between tables.
  DelJoin      Removes an existing join between tables.
  NextTable    If there are more tables than can fit on the screen, moves the
               cursor to the first table not currently displayed.
  TableDef     Displays details about the structure of the current table.
  Utilities    Displays a selection of utility functions.
  ZoomOut      Displays the Visual Query in compressed mode.
  Help         Displays this help screen.
  Cancel       Cancels your changes and displays the Application Flow Diagram.
  End          Saves any changes to the Visual Query and displays the
               Application Flow Diagram.

  Keys(SH-F1)  SubTopics(SH-F2)  Help(F1)  End(F10)
```

When you select SubTopics from the menu on the preceding screen, a pop-up window is displayed containing the subtopics specified in the help file, as shown in the following figure:



```
Ingres - Vision
HELP -- Visual Query

 Choose a SubTopic     shows the structure of the query (frame definition)
                       To change the frame definition, edit the entries on
 Editing Append frames menu operations.  Subtopics explain the Visual Query
 Editing Browse frames l.
 Editing Update frames okup table to the Visual Query.
 Joins                  Lookup table from the Visual Query.
 Lookup tables         edit the form for a frame.
 Screen layout         w join between tables.
 Scrolling             n existing join between tables.
                       are more tables than can fit on the screen, moves the
               cursor to the first table not currently displayed.
  TableDef     Displays details about the structure of the current table.
  Utilities    Displays a selection of utility functions.
  ZoomOut      Displays the Visual Query in compressed mode.
  Help         Displays this help screen.
  Cancel       Cancels your changes and displays the Application Flow Diagram.
  End          Saves any changes to the Visual Query and displays the
               Application Flow Diagram.

  Select(F9)  Cancel(F7)  Help(F1)
```

help_frs Statement—Include the Ingres Help Facility

This statement obtains access to the Ingres help facility.

Syntax

```
help_frs (subject = subjectname, file = filename)
```

Description

The help_frs statement enables you to include a help facility in an application. The Ingres help facility provides the user with access to information on the current form, the fields of the form, field validation criteria, and the current function/control/arrow key mappings.

Filename is the name of the file containing the help text for the current form. If the application is to be run by different users specify the full directory specification. The text can be different for each form in the application. The help text is displayed in a table field format. If the file specified by *filename* does not exist, a message telling the user that no help exists for that *subjectname* appears on the screen. Both *filename* and *subjectname* must be expressed as quoted string literals or program string variables.

During the development of a forms-based system that uses the Ingres help facility, it is often useful to escape to a system text editor to edit the help text while running the application. The logical/environment variable II_HELP_EDIT provides this capability. By setting this name at the operating system level to any value, such as TRUE, you cause an extra menu operation, Edit, to appear under the WhatToDo operation in the help facility. Selecting the Edit operation invokes the default system text editor on the current frame's help file. After you have saved the file the new help file constitutes the help text for the frame. You can also change the default text editor by setting ING_EDIT to the name of your preferred editor.

Example—help_frs statement:

Example 1:

Provide the Ingres help facility in the personnel form, in the current directory.

```
exec frs activate menuitem 'Help';
 exec frs begin;
    exec frs help_frs (subject = 'Personnel form',
      file = 'personnel.hlp');
exec frs end;
```

By selecting the menu item Help, the user gains access to the Ingres help facility. The user can obtain information on the current form using the text contained in the named file. If such a file does not exist, a message stating that the help file for Personnel form cannot be opened appears at the top of the screen. In addition, the user can look up field validations and function/control/arrow key mappings.

Initialize Statement—Initialize a Form

This statement performs any necessary initialization on the form.

Syntax

```
initialize [(fieldname = value
     {, fieldname = value})]
[begin
     initialization code;
end]
```

Description

The initialize statement lets you transfer data into the form at the start of a display loop and perform any necessary initial operations for the form. This statement is executed when the display loop begins, before the form actually appears on screen.

The list of *fieldnames* identifies the fields into which values are assigned. The fields listed must be simple fields. (You can transfer data into table field columns by means of the loadtable, insertrow, and putrow statements.) You can specify *fieldname* using a quoted or unquoted character string or a program variable. Similarly, *value* can be either a literal or a variable. A field and the value assigned to it must have compatible data types. (See your host language companion guide for information about compatible data types.)

There are two ways to insert a null into a field. First, you can specify *value* as the key word null. This method assigns a null to the field whenever the statement executes. The alternate, and more flexible method, uses an indicator variable. Using an indicator variable allows the user or program to decide at run time whether to place a null in the field.

An indicator variable is a two-byte integer variable associated with the variable used to assign values into the field. It is specified using the syntax:

```
initialize (fieldname = var:indicator_var)
```

You can only use an indicator variable when you use a variable for *value*. Also, you must have previously declared the indicator variable in a host variable declaration section.

When an indicator variable is set to -1, any value in its associated variable is ignored and a null is assigned to the specified field. You must set the indicator variable before executing the initialize statement and the receiving field must be nullable. See your host language companion guide for a complete description of the use of indicator variables.

If you include the optional begin/end block of code, you cannot place any other statements or program comment lines between the initialize statement and the block. However, the block itself can contain any host language or embedded SQL statement, as well as comment lines.

When used, you must put the initialize statement in a display block. Although not required, the initialize statement is often useful as a way to mark the start of the display loop.

Examples—initialize statement:

Example 1:

Place data from program variables into a form at the start of its display.

```
exec frs initialize (ename = :namevar,
                     sal = :salvar);
```

Example 2:

Place data in the form and display a message.

```
exec frs initialize (ename = 'Sally',
                     sal = 30000.00);
exec frs begin;
     exec frs message 'You can begin editing data.';
    exec frs sleep 2;
exec frs end;
```

Example 3:

Place a **null** into a field.

```
    exec frs initialize (spouse = null);
```

Example 4:

Place a value into a nullable field.

```
exec frs initialize
               (spouse = :spouse_var:indicator_var);
```

Inittable Statement—Initialize a Table Field

This statement initializes a table field, associating it with a data set.

Syntax

```
inittable formname tablename [tablemode]
    [(columnname = format {, columnname = format})]
```

Description

The inittable statement initializes a table field by linking it with a data set, specifying its mode, and defining any hidden columns.

A data set is the internal data structure that holds rows of values for the table field. An initialized table field can have many rows in its data set, even though not all can be displayed. For a full discussion of table fields and data sets, see Table Fields (see page 579).

Formname specifies the name of the form in which the table field identified by *tablename* is displayed. You can express either name as a quoted or unquoted character string or as a program variable.

The *tablemode* sets the table field's display mode. It must be a quoted or unquoted character string or program variable that evaluates to one of the following values:

read

In this mode, the user can only browse the contents of the table field by scrolling through the values in the data set. This mode is normally used with table fields that merely display data or when all changes to the data are made by means of menu operations.

update

In this mode, the user can scroll through the values in the data set and change the data displayed. However, the user cannot add rows to the end of the data set. This mode is useful in applications that display a series of records for possible modifications by the user.

fill

This mode is similar to the update mode, except that it also allows the user to enter new rows at the end of the table field. Rows so entered are automatically recorded in the data set. Any column into which the user does not enter data is given a default value: 0 (zero) for numeric (float, float4, integer, smallint, integer1, and money) columns, and blanks for string (char, varchar, c, text, and date) columns. However, if the column is of a nullable data type, the default value is always null.

query

This mode has all the capabilities of the fill mode, but also allows the user to enter comparison operators (for example, <> or >=,) which can be retrieved into program variables. Query mode is useful for allowing a user to specify parameters for a database retrieval.

The mode of the table field is normally independent of the mode of its form. However, if the form is displayed in read mode, then the table field behaves as if it is in read mode also. The real mode of the table has not been lost, however.

The default mode is **fill**.

The optional column list is used to define hidden columns for the table field. Hidden columns are columns of data that are not displayed on the form and are therefore invisible and inaccessible to the user, but which can be accessed by the program in the same way the program accesses displayed columns. A *columnname* can be any valid Ingres name of up to 32 bytes and can be expressed as a quoted or unquoted character string literal or a program variable. The column's *format* defines its data type and length. The *format* can be any legal Ingres data type. See your query language reference guide for a complete list of legal data types.

The format can include the with null clause to specify that the hidden column is nullable or the not null clause to specify a non-nullable hidden column. Not null implies not null with default. By default, a hidden column is non-nullable.

For greater runtime flexibility, you can put the complete list of column names and formats in a single program string variable.

The inittable statement can appear at any point after the form that contains the table field has been declared. However, because a table field cannot have a data set associated with it until inittable is performed, most applications execute the inittable statement before displaying the form containing the table field.

If you issue the inittable statement twice for the same table field, the second execution eliminates the previous values in the data set. If the second execution specifies the same mode as the first, the second execution is functionally equivalent to the clear field statement.

Examples—inittable statement:

Example 1:

Initialize a table field in the read mode with no hidden columns.

```
exec frs inittable empform employee read;
```

Example 2:

Initialize a table field in the default mode and specify two hidden columns for it.

```
exec frs inittable empform employee
    (sal = float4, eno = integer);
```

Example 3:

If the program is in supervisor mode, initialize the employee table field in update mode. Otherwise, initialize it in read mode.

```
if (supervisor_mode) then
    mode = 'update';
else
    mode = 'read';
end if;
exec frs inittable empform employee :mode
    (eno = integer);
```

Example 4:

Initialize a table field with nullable hidden column.

```
exec frs inittable empform employee
    (spouse = varchar(40) with null);
```

Example 5:

Initialize a table field with a non-nullable hidden column.

```
exec frs inittable empform employee
    (salary = money not null);
```

inquire_frs Statement—Retrieve FRS Runtime Information

This statement provides runtime information concerning the FRS.

Syntax

```
inquire_frs object_type {parent_name} [row_number]
     (variable = frs_constant[(object_name)]
     {, variable = frs_constant[(object_name)]})
```

Description

The inquire_frs statement retrieves a wide range of information about a variety of form objects or the FRS itself. For example, you can determine the type of terminal in use, the display mode of a form, the validation string associated with a column in a table field, or the screen coordinates of a field, a pop-up form, or the cursor. This statement is particularly useful because FRS status and error information is not reflected in the SQLCA and, consequently, you cannot use the embedded SQL whenever statement to handle forms program errors. The inquire_frs statement is essential for determining the effect of forms statements.

To obtain information about all of a form's fields in sequence or all of the columns in a table field, you can use the inquire_frs statement in conjunction with the formdata or tabledata statements, respectively. See the statement descriptions for those statements for more information.

The syntactical elements have the following meanings:

- *object_type* is the type of object about which you are requesting information. Specify this as a string, with or without quotes. The following table lists valid object types.

Object Type	**Information Returned**
Frs	Information about the FRS
Form	Information about a form
Field	Information about a field in a form
Table	Information about a table field
Column	Information about a column in a table field
Menu	Information about the menu for the current display loop
Row	Information about a row in a table field

- *parent_name* identifies the form and, when necessary, the table field which contains the specified *object_name*. Specify the *parent_name* as a quoted or unquoted string literal or as a program variable. If *parent_name* is a quoted blank or empty string, the current parent is used. You cannot refer to more than one *parent_name* in a single inquire_frs statement. Use the following list for reference, keeping in mind that *object_name* must be of the type specified by *object_type*:

Object Type	**Parent Name**
Frs	None
form	None
field	formname
table	formname
column	formname tablename
menu	*formname*
row	formname tablename

- *row_number* identifies a specific row in a table field display and is only accepted when the *object_type* is **row**. You can use an integer literal or integer variable to represent *row_number*. If the specified table field is linked to a data set, then the row number must refer to a row in the display that contains data. For example, if the table field was defined, in VIFRED, to display four rows but only the first two rows have data in them, specify only the values 1 or 2 for *row_number*. If the table field has not been initialized, specify the values 1 through 4, inclusive, for *row_number*. If you do not include a row number, then the current row is assumed, that is, the row on which the cursor rests.
- *variable* is the name of a program variable into which the information is to be retrieved. The data type of each variable must be appropriate to the type of information which it receives.
- *frs_constant* represents a key word that indicates what type of information is wanted. Each type of object has a variety of *frs_constants* that provide a variety of information about the specified object. See the detailed descriptions that follow for each object type.

- *object_name* is the name of the specific object about which you are inquiring. You can express this as a quoted or unquoted string literal or, if the *object_type* is **form**, as a program variable. A single **inquire_frs** statement can contain references to more than one object, however, all of the objects referenced must have the same parents. For example, if the *parent_name* is form1, *object_name* can refer to any field on form1, but you cannot refer to fields that appear on form2 or form3, and so forth. Similarly, if the *parent_names* identify a table field, then the object names must be names of columns in that table field. You cannot reference columns in other table fields or other simple fields on the form.

If you do not include an object name, assumes the current object is assumed, that is, the object on which the cursor rests. However, when the FRS itself is the *object_type*, not all of its *frs_constants* require an object name.

The object names for the various object types are listed in the following table:

Object Type	Object Names
Frs	frs_object (varies, according to FRS constant)
form	Formname
field	Fieldname
table	Tablename
column	Columnname
row	Columnname

The following sections describe the use of the inquire_frs statement for each of the object types described above.

Retrieving Information about the Runtime System

The frs object name is used to retrieve information about the FRS. The syntax of this statement is:

```
inquire_frs frs
    (variable = frs_constant[(frs_object)]
    {, variable = frs_constant[(frs_object)]})
```

The following table lists the accepted *frs_constants*, their associated *frs_objects*, and the data type of the returned value:

Frs_constant	Data Type	Returns
activate	integer	Returns 1 if the type of field activation specified by its

Frs_constant	Data Type	Returns
		frs_object is turned on; otherwise, it returns 0. Valid frs_objects are: **before:** Returns 1 if field entry activation is enabled. **nextfield:** Returns 1 if field exit activation is enabled when the user moves forward out of a field. If an exit activation is defined for a field, the activation occurs when the user moves forward out of the field. You must use the **set_frs** statement to enable activations arising from a backward exit. **previousfield**: Returns 1 if field activation occurs when the user moves backward to the previous field on a form. **menu**: Returns 1 if field activation occurs when the user presses the Menu key. **keys**: Returns 1 if field activation occurs when the user presses a function, arrow, or control key associated with an FRS key. **menuitem**: Returns 1 if field activation occurs when the user selects a menu item (or a function, arrow, or control key mapped to a menu item). You can use the set_frs statement to turn on and off the activations for the activities represented by the frs_object. A field activation only takes place under the above conditions if an activate field statement has been specified for the field where the cursor is resting or entering.
columns	integer	Returns the size of the terminal screen, in terms of the number of columns. Omit frs_object.
command	integer	Returns an integer representing the last FRS command entered by the user. An application can use this information inside an activation block when it needs to know exactly which command caused an activation to occur**.** Omit frs_object. Returned integer values and their meanings are: **Value** **FRS Command** **0** Undefined **1** menu key **2** any FRS key **3** any menu item **4** next field or autoduplicate previousfield **5** downline **6** upline **7** newrow **8** clearrest

Frs_constant	Data Type	Returns
		9 scrollup **10** scrolldown **11** nextitem or auto tab **12** timeout **13** new field selected by mouse **14** **Undefined** (0) is returned in two instances: If an inquire is executed after exiting from a submenu block and before reentering the display loop, If the return from a prompt or message was not caused by a timeout.
cursorcolumn	integer	Returns the current column position of the cursor. Omit frs_object.
cursorrow	integer	Returns the current row position of the cursor. Omit frs_object.
editor	integer	Returns 1 if the FRS editor command is enabled; 0 if it is not. Editor does not accept an frs_object.
errorno	integer	Returns the FRS error number; set if an error occurs when executing a forms statement in the current display loop. Errorno does not accept an frs_object.
Errortext	character	Returns the text of error message associated with the error number returned by errorno. Returns an empty string if there is no current error. Errortext does not accept an frs_object.
getmessages	integer	Returns 1 if the error messages associated with the getform, getrow, and unloadtable statements are suppressed. If they are not, this returns 0. Does not take an frs_object.
label	character	Returns the alias given to the control, function, or arrow key to which the specified frs_object is mapped. If an alias is defined for a control key, arrow key, or function key, the alias is displayed rather than the key's name in the help facility display and on the menu line for menu items that are mapped to that key. Aliases are very useful for determining which key to press to execute a desired function. By using the letters on a terminal key cap as aliases for function keys, you can make it easier for the user to select the correct key. You can define an alias in the mapping file or using the set_frs statement. See the *Character-based Querying and Reporting Tools User Guide* for details on the mapping file. The following are valid frs_objects:

Frs_constant	Data Type	Returns
		menuN: Returns the alias for the key to which the Nth menu item on the menu line is mapped. N must be in the range of 1 to 25, inclusive. **frs_command**: Returns the alias for the key to which the specified frs_command is mapped. FRS commands are listed in *Character-based Querying and Reporting Tools User Guide*. **frskeyN**: Returns the alias for the key to which the Nth frskey is mapped. N must be in the range of 1 to 40, inclusive. Inquire_frs returns a null string if the specified frs_object is not mapped to a control, function, or arrow key. If the object is mapped to a key but no alias has been defined for that key, then inquire_frs returns the name of the key. Control keys are returned as Control X where X can be any one of the characters A-Z or Esc or Del. Function keys are returned as PFN where N is a number from 1 to 40.
last_frskey	integer	Returns the number of the FRS key selected by the user. The numbers returned start at 1. If the activation was not the result of an FRS key selection, returns a -1. Useful when two or more FRS keys are combined in a single activation block and the program must know which key was actually selected. Last_frskey does not take an frs_object.
Map	character	Returns the name of the control, functions, or arrow key to which the specified frs_object is mapped. Returns a null string if there is no key mapped to the object. (See *Character-based Querying and Reporting Tools User Guide* for information about key mapping.) The following are valid frs_objects: **menuN**: Returns the key to which the Nth menu item on the menu line is mapped. N must be in the range of 1 to 25, inclusive. **frs_command**: Returns the key to which the specified frs_command is mapped. FRS commands are listed in *Character-based Querying and Reporting Tools User Guide*. **frskeyN**: Returns the key to which the Nth frskey is mapped. N must be in the range of 1 to 40, inclusive.
mapfile	character	Returns the file specification for the application's FRS key mapping file. Mapfile does not accept an frs_object.
menumap	integer	Returns 1 if the menu line currently displays association between menu items and function, control, or arrow keys, or 0 otherwise. Menumap does not accept an frs_object.

Frs_constant	Data Type	Returns
outofdatamessage	integer	Returns 0 if the Out of Data message is suppressed, 1 if the message is displayed, or 2 if the FRS sounds the terminal bell when the user attempts to scroll off the top or bottom of a table field.
rows	integer	Returns the size of the terminal screen, in terms of the number of rows. Omit frs_object.
shell	integer	Returns 1 if the FRS shell command is enabled; 0 if it is not. Shell does not accept an frs_object.
terminal	character	Returns the type of terminal, such as vt100.Terminal does not accept an frs_object.
timeout	integer	Returns the number of seconds specified to wait before a timeout occurs. Timeout does not use an frs_object.
validate	integer	Returns a 1 if the type of field validation specified by its frs_object is turned on; otherwise, it returns 0. Valid frsobjects are: **nextfield**—Returns 1 if field validation occurs when the user moves forward to the next field on a form. **previousfield**—Returns 1 if field validation occurs when the user moves backward to the previous field on a form. **menu**—Returns 1 if field validation occurs when the user presses the Menu key. **keys**—Returns 1 if field validation occurs when the user presses a function, arrow, or control key associated with an FRS key. **menuitem**—Returns 1 if field validation occurs when the user selects a menu item (or a function, arrow, or control key mapped to a menu item). You can use the set_frs statement to turn on and off the validations for the activities represented by the frs_objects.

Retrieving Information about a Form

The form object type is used to obtain information about a form. The specified form must have been declared in the program before you can reference it in an inquire_frs statement.

The syntax is:

```
inquire_frs form
    (variable = frs_constant[(formname)]
    {,variable = frs_constant[(formname)]})
```

The *formname* identifies the form about which you are inquiring. If you do not specify a form, the current form is assumed, that is, the form on which the cursor currently rests. (The name constant always refers to the current form and, consequently, never needs a specified form name.) You can use a quoted or unquoted string literal or a program variable to represent *formname*.

Valid values for *frs_constant* are listed in the following table:

Frs_constant	Data Type	Returns
change	integer	Returns 1 if the user has changed any displayed data on the form, or 0 otherwise. This constant is set by a user typing at the keyboard. The FRS command clearrest, which is mapped to the Return key on many terminals, sets this constant. Program-based changes to the screen, using putform and so forth, do not affect change. This constant can also be set or altered with the set_frs statement.
columns	integer	Returns the form's size, expressed as the number of columns. If the form is a pop-up, this includes the columns occupied by the form's border.
exists	integer	Returns 1 if the specified form exists, 0 if the form does not exist.
field	character	Returns the name of the current field on the form.
mode	character	Returns the form's display mode. Returned values are one of the following: **none** **update** **fill** **query** **read** **formdata** None is returned if the specified form is not

Frs_constant	Data Type	Returns
		displayed. Formdata is returned if the inquire statement is issued inside a formdata loop.
name	character	Returns the name of the form. Because this constant always refers to the current form, it is not necessary to specify a formname.
rows	integer	Returns the form's size, expressed as the number of rows. If the form is a pop-up, the number includes the rows occupied by the form's border.

Retrieving Information about a Field

The field object type retrieves information about a field in a form. The syntax is:

```
inquire_frs field formname
     (variable = frs_constant[(fieldname)]
     {,variable = frs_constant[(fieldname)]})
```

Formname identifies the form that contains the specified field. If *formname* is left as an empty or blank string and no *fieldname* is specified after the constant, then the constant references the current field of the current form. This format is particularly useful in a form display loop or the formdata loop. You can express both formname and fieldname as quoted or unquoted string literals or program variables.

The *fieldname* can identify either simple fields or table fields; some inquiries (for example, datatype) are meaningless when applied to tablefields as a whole. To retrieve information about a table field or its columns, see Retrieving Information about a Table Field (see page 684) and Retrieving Information About a Column (see page 685).

Valid values for *frs_constant* are listed in the following table:

Frs_constant	Data Type	Returns
change	integer	Returns1 if the user has typed into the field, 0 otherwise. A corresponding set_frs statement is available to set the change variable for the field. A field's change variable is set to 1 whenever a runtime user types into the field. The clearrest FRS command, mapped to the Return key on many terminals, sets this constant. It is set to 0 at the start of a display loop, when a value is placed into the field by a putform statement or when the field is cleared by a clear statement. You cannot specify a table field when

Frs_constant	**Data Type**	**Returns**
		you use this constant.
color	integer	Returns the color code (0-7) for the field. The default color is 0.
columns	integer	Returns the number of columns occupied by the current simple field input area. *Formname* must be a quoted empty or blank string, and fieldname is not specified.
datatype	integer	Returns the true data type of field: **date** **3** **money** **5** **decimal** **10** **char** **20** **varchar** **21** **integer** **30** **float** **31** **c** **32** **text** **37** Nullable data types are returned as the negative value of the non-nullable ones. For example, a nullable integer data type returns -30.
derived	integer	Returns a 1 if the field is a derived field or 0 if it is not.
derivation_string	character	Returns the derivation string of the field. If the field is not a derived field, this returns an empty string.
display	integer	Returns a 1 if the attribute specified by *display* is in effect for the field or 0 if it is not. *Display* can be any of the following constants: **reverse** **blink** **underline** **intensity** **normal** **displayonly** **invisible** Exception: normal returns 1 if no attributes are in effect for the field or 0 if any attribute is in effect; the settings of the displayonly and invisible attributes do not affect the value returned for the normal constant. To set display attributes, use the set_frs statement.
exists	integer	Returns 1 if the specified field exists, 0 if the field

Frs_constant	Data Type	Returns
		does not exist. Valid for simple fields and table fields.
format	character	Returns the format string specified for field.
inputmasking	integer	Returns 1 if inputmasking is in effect, 0 if it is not.
length	integer	Returns the length, in bytes, of the data area in the field.
mode	character	Returns the field's display mode as specified: fill, update (table field only), query or read. The specified field can be either a simple field or a table field.
name	character	Returns the name of the field. Because this constant always refers to the current field, it is not necessary to specify a *fieldname*.
number	integer	Returns the sequence number within the form for the field (sequence numbers start at 1). (If your form was created using Ingres 6.3 or an earlier release of Ingres, then display-only fields return a negative number.)
rows	integer	Returns the number of rows occupied by the current simple field input area. *Formname* must be a quoted empty or blank string, and fieldname is not specified.
startcolumn	integer	Returns the column position of the upper left hand corner of the current simple field input area. This constant does not allow the use of an explicit *fieldname*, and *formname* must be a quoted empty or blank string. The returned values are relative to the terminal screen, whose origin point is its upper left hand corner, described as row 1, column 1. If the value returned is zero or negative, then the upper left hand corner of the field is not visible on the screen.
startrow	integer	Returns the row position of the upper left hand corner of the current simple field input area. This constant does not allow the use of an explicit *fieldname*, and *formname* must be a quoted empty or blank string. The returned values are relative to the terminal screen, whose origin point is its upper left hand corner, described as row 1, column 1. If the value returned is zero or negative, then the upper left hand corner of the field is not visible on the screen.
table	integer	Returns 1 if this field is a table field, or 0 if it is a regular field.
type	integer	Returns the basic data type of field (1=**integer**, 2=**float**, **money** or **decimal**, 3=**varchar** or **char** or **c**

Frs_constant	Data Type	Returns
		or **text** or **date**).
valid	character	Returns the validation string for field.

Retrieving Information about a Table Field

The table object type retrieves information about a table field. The syntax is:

```
inquire_frs table formname
     (variable = frs_constant[(tablename)]
     {,variable = frs_constant[(tablename)]})
```

Formname identifies the form in which the table field is displayed. If *formname* is left as an empty or blank string and *tablename* is not specified after the constant, then the constant refers to the current table field in the current form. This form of the statement is particularly useful while displaying a table field or while in the **f**ormdata loop. You can express both formname and tablename as quoted or unquoted string literals or as program variables.

Note: To determine if a specific table field exists, use the inquire_frs field (exists) option, described in the section entitled Retrieving Information About a Field. To determine if a particular column in a table field exists, use the inquire_frs column (exists) option, described in Retrieving Information About a Column (see page 685).

The legal values for *frs_constant* are listed in the following table:

Frs_constant	Data Type	Returns
column	character	Returns the name of column on which the cursor is positioned within the table field.
datarows	integer	Returns the number of non-deleted rows stored in the data set of the table field. If the table field is not linked to a data set, then this is equivalent to lastrow.
lastrow	integer	Returns the number of displayed rows on the table field that actually contains data.
maxcol	integer	Returns the number of displayed columns in the table field as defined in VIFRED, including any currently invisible columns.
maxrow	integer	Returns the number of displayed rows within the table field, as defined in VIFRED.
mode	character	Returns the table field initialization mode (fill, update, query, or read).

name	character	Returns the name of the table. Because this constant always refers to the current table field, it is not necessary to specify a *tablename*.
rowno	integer	Returns the current row number within display (displayed rows are numbered from 1).

Retrieving Information about a Column

The column object type is used to obtain information about a column in a table field. The syntax is:

```
inquire_frs column formname tablename
     (variable = frs_constant[(columnname)]
     {,variable = frs_constant[(columnname)]})
```

Formname identifies the form that contains the specified table field, and *columnname* identifies the column in the specified table field. If *formname* and *tablename* are left as null strings and *columnname* is not specified after the constant, then the column on which the cursor is currently resting is assumed. This format is particularly useful within the tabledata loop. You can use quoted or unquoted string literals or program variables to represent *formname*, *tablename*, and *columnname*.

Valid values for *frs_constant* are listed in the following table:

Frs_constant	Data Type	Returns
color	integer	Returns the color code (0-7) for the column. The default color is 0.
columns	integer	Returns the width of the table field column as defined by its display format. *Formname* and *tablename* must be quoted blank or empty strings.
datatype	integer	Returns the true data type of field: **date** **3** **money** **5** **decimal** **10** **char** **20** **varchar** **21** **integer** **30** **float** **31** **c** **32** **text** **37**
		Nullable data types are returned as the negative value of the non-nullable ones. For example, a nullable integer data type

Frs_constant	Data Type	Returns
		returns -30.
Derivation_string	character	Returns the derivation string for the specified column. If the column is not a derived field, then this returns an empty string.
derived	integer	Returns 1 if the specified column is a derived field, or 0 if it is not.
display	integer	Returns 1 if the specified display attribute is on for the column or a 0 if it is not. *Display* can be any of the following constants: **reverse** **blink** **underline** **intensity** **normal** **displayonly** **invisible** Normal returns 1 if no attributes are in effect for the column or 0 if any attribute is in effect; the settings of the displayonly and invisible attributes do not affect the value returned for the normal constant. To set the display attributes for columns, use the set_frs statement.
exists	integer	Returns 1 if the specified column exists, 0 if the column does not exist.
format	character	Returns the format string specified for the column.
length	integer	Returns the length, in bytes, of the data area in the column.
mode	character	Returns the mode of display for the column as specified (fill, query or read).
name	character	Returns the name of the column. Because this constant always refers to the current table field column, it is not necessary to specify a columnname.
number	integer	Returns the column's sequence number within the table (numbering begins at 1).
rows	integer	Returns the number of lines defined by the display format of the column. Formname and tablename must be quoted blank or empty strings.
type	integer	Returns the basic type of column (1=integer, 2=float or money or decimal, 3=varchar or char or c or text or date).
valid	character	Returns the validation string defined for the column.

Retrieving Information About a Row in a Table Field

The **row** object type can be used to retrieve information about a row within a table field. The syntax is:

```
inquire_frs row formname tablename [row_number]
    (variable = frs_constant[(columnname)]
     {, variable = frs_constant [(columnname)]});
```

Formname identifies the form which contains the specified table field, and *columnname* identifies the column in the table field. If *formname* and *tablename* are left as empty strings and no *columnname* is specified after a constant, the table field cell on which the cursor is currently resting is assumed.

Valid values for *frs_constant* are listed in the following table:

Frs_constant	Data Type	Comment
change	integer	Returns to 1 if the user has typed into the column, 0 otherwise. To set the change variable for the column, use set_frs. A field's change variable is set to 1 when the runtime user types into the column. It is set to 0 at the start of a display loop, when a value is placed into the column by a putrow statement, when the column is cleared by a clear statement, or when a new row is created as with an insertrow statement. To inquire about a particular displayed row, specify *row_number;* to specify the current row, omit *row_number.* In an unloadtable statement loop you cannot specify row_number, and therefore can only inquire on the row just unloaded.
display	integer	Returns 1 if the specified display attribute is turned on for the specified cell (row and column). For color, returns color code (0 - 7) *Display* must be one of the following: **blink** **color** **intensity** **normal** **reverse** **underline**
startrow	integer	The starting row coordinate for the current input cell of a table field. This coordinate is relative to the terminal screen coordinates. The terminal screen's origin is its upper left hand corner, described as 1,1. If the value returned by startrow is zero or negative, then the input cell's starting row is off of the screen. A zero is also returned if the inquire_frs statement is issued from an

Frs_constant	Data Type	Comment
		activate scroll up/down block.
startcolumn	integer	The starting column coordinate for the current input cell of a table field. This coordinate is relative to the terminal screen coordinates. The terminal screen's origin is its upper left hand corner, described as 1,1. If the value returned by startcolumn is zero or negative, then the input cell's starting column is off of the screen. A zero is also returned if the inquire_frs statement is issued from an activate scroll up/down block.

To determine if a display attribute is turned off or on for a particular value in the table field data set, you must use the inquire_frs statement within an unloadtable loop. Omit the *row_number* parameter: statements within an unloadtable loop always operate on the current row.

Retrieving Information about Menu Items

The menu object type can be used to retrieve information about menu items in the menu line for the current display loop. The syntax is:

```
inquire_frs menu formname | empty_string
    (variable = frs_constant[(menu_item)]
```

If you are referring to the current form you can omit *formname* and specify an empty string instead (in ESQL, specify ' ', and in EQUEL specify " "). The formname parameter can be specified to access menu objects in other display loops. The *menu_item* parameter must specify the name of the menu item about which you are inquiring.

The legal values for *frs_constant* are listed in the following table:

Frs_constant	Data Type	Comment
Active	Integer	Returns 1 if the specified menu item is active (displayed and available to the user), 0 if the menu item is inactive. To enable or disable menu items, use the set_frs statement (with the active option).

Examples—inquire_frs statement:

Example 1:

Find out if an error occurred. If so, call a clean-up routine.

```
exec frs inquire_frs frs (:err = errorno);
 if (err < 0) then
    call clean_up(err);
end if;
```

Example 2:

Confirm that user changed some data on the currently displayed form before updating the database.

```
exec frs activate menuitem 'Update' (validate = 1);
 exec frs begin;
    exec frs inquire_frs form (:updated = change);
    if (updated = 1) then
        exec frs getform (:newvalue = value);
        exec sql update newtable
            set newvalue = :newvalue
            where .....;
    end if;
exec frs end;
```

Example 3:

Find out the mode and current field of a form whose name is passed as a parameter.

```
exec frs inquire_frs form
    (:mode = mode(:formname),
               :fldname = field(:formname));
```

Example 4:

Implement a generalized help facility based on the current field name.

```
exec frs activate menuitem 'HelpOnField';
 exec frs begin;
    exec frs inquire_frs form (:fldname = field);
    Place appropriate file for "fldname" into
      "filebuf";
    exec frs helpfile 'Field Help' into :filebuf:
exec frs end;
```

Example 5:

Find out if the current field in form empform is a table field before issuing the deleterow statement on the current row.

```
exec frs activate menuitem 'Deleterow';
 exec frs begin;
    exec frs inquire_frs field empform
        (:fldname = name, :istable = table);
    if (istable = 0) then
        exec frs message
           'You must be on the table field to delete
           the current row.';
        exec frs sleep 2;
    else
        exec frs deleterow empform :fldname;
    end if;
 exec frs end;
```

Example 6:

Allow the runtime user to change the row following the current row in a table field. Verify that the current field is a table field and that the next row of the table field is visible.

```
exec frs activate menuitem 'ChangeNextRow';
 exec frs begin;
    exec frs inquire_frs field '' (:istable = table);
    if (istable = 0) then
        exec frs message 'You must move to the
            table field';
        exec frs sleep 2;
    else
        inquire_frs table ''
            (:fldname = name, :currow = rowno,
             :lastrow = lastrow);

        if (currow = lastrow) then
            exec frs message 'You must scroll
                                 in a new row';
            sleep 2;
        else
            currow = currow + 1;
            /*
            ** Update data in row specified by
            ** 'currow'
            */
       end if;
    end if;
exec frs end;
```

Example 7:

Inquire whether a field was changed by the runtime user.

```
exec frs activate menuitem 'Salary';
 exec frs begin;
    exec frs inquire_frs field (:changed =
                               change(salary));
    if (changed = 1) then
        log salary change for employee;
        exec frs set_frs field (change(salary) = 0);
        /* clear the change variable */
    end if;
exec frs end;
```

Example 8:

Check to see if a change was made to make the application more efficient. For the example below, assume that the only field on the form is a table field with columns name and rank.

```
exec frs activate menuitem 'Update';
 exec frs begin;
    /*
    ** Check if a change was made to column "rank"
    ** in the current row.
    */
    exec frs inquire_frs row
        (:changed = change(rank));
    /* Only need to update database
    ** if a change was made.
    */
    if (changed = 1) then
        get information and update database;
        exec frs set_frs row (change(rank) = 0);
        /* clear the change variable */
    end if;
 exec frs end;
```

Example 9:

Only validate field key if value has changed:

```
exec frs activate field key;
 exec frs begin;
    exec frs inquire_frs field
        (:changed = change(key));
     if (changed = 1) then
        perform field validation
        /* clear the change variable */
        exec frs set_frs field (change(key) = 0);
     endif;
exec frs end;
```

insertrow Statement—Insert a Row

This statement inserts a new row into a table field.

Syntax

Non-dynamic version:

```
insertrow formname tablename [row]
    (columnname = value [{, columnname = value}])
    [with (attribute(columnname) = 0 | 1 | color
    {, attribute(columnname) = 0 | 1 | color})]
```

Dynamic version:

```
insertrow formname tablename [row]
    using [descriptor] descriptor_name
```

Description

The insertrow statement inserts a new row into the table field display. Unlike other methods of adding rows to the data set, the insertrow statement allows you to add rows to the beginning, middle, or end of the data set. However, insertrow can only insert rows into the displayed table field; it cannot insert rows into any part of the data set not currently visible on the form. For example, if you want to insert a row at the beginning of your data set, the beginning of the data set must be displayed in the table field when you issue the insertrow statement.

The insertrow statement also allows you to specify the row state of the newly inserted row (see below).

The *formname* is the name of the form in which the table field *tablename* is displayed. You can express both *formname* and *tablename* as either quoted or unquoted string literals or program variables.

Row identifies one of the rows in the table field display. If *row* is specified, the new row is inserted after the specified *row*. For example, if your table field display contains five rows, then *row* can be any value from 0 to 5. If you insert a row, specifying a value of 3 for *row*, then the new row is inserted after the currently displayed row 3.

Choosing a value of 0 for *row* inserts the new row as the top (first) row in the table field display.

If the table field has been initialized, the value you choose for *row* cannot be greater than the number of displayed rows or smaller than zero. This means that regardless of how many rows the table field is capable of displaying, *row* can identify only a row that is currently displayed in the table field. For example, if your table field display has the capacity to display five rows, but has only two rows loaded into it, then the row number must be either 0, 1, or 2. If *row* is omitted, the new row is inserted after the row on which the screen cursor is currently positioned.

When a row is inserted into a table field with a data set, all rows below the inserted row are scrolled down. If the row is inserted after the last displayed row (for instance, if the table field can display four rows and *row* is 4), the rows above it are scrolled up, and the new row becomes the last displayed row.

The non-dynamic insertrow statement assigns values directly to columns in the table field. The list of *columnnames* identifies the columns receiving values and the values they are to receive. You can assign values to hidden, invisible, or displayed column in the table field; you cannot assign values to derived columns. The data type of the *value* must be compatible with the data type of the column. You can specify *columnname* using a quoted or unquoted character string literal or a program variable.

Only the specified columns are assigned values. Omitted columns are assigned either nulls, if they are nullable, or default values (blanks for character columns and zeros for numeric columns). If you omit the column list, the inserted row is blank.

By default, rows inserted by the insertrow statement have a state of UNCHANGED. If the table field has an associated data set, you can assign the state of the new row by specifying _state=*value* in the column list. For example, you can specify an initial row state of UNDEFINED (0) when the row is inserted; the state is set to NEW if the user types into the row.

If you want to insert a null, there are two ways to do so. First, you can specify the column's associated *value* as the key word null. This method assigns a null to the column whenever the statement executes. The alternate, and more flexible method, uses an indicator variable. Using an indicator variable allows the user or program to decide at run time whether to place a null in the column.

An indicator variable is a two-byte integer variable associated with the variable used to assign values into the column. It is specified using the syntax:

```
insertrow (columnname = var:indicator_var)
```

You can only use an indicator variable when you use a variable for *value*. Also, you must have previously declared the indicator variable in a host variable declaration section.

When an indicator variable is set to -1, any value in its associated variable is ignored and a null is assigned to the specified column. You must set the indicator variable before executing the insertrow statement and the receiving column must be nullable. See your host language companion guide for a complete description of the use of indicator variables.

If the *columnname* is the constant _state, *value* must evaluate to one of the following:

0 — UNDEFINED
1 — NEW
2 — UNCHANGED
3 — CHANGED

You must not assign values to any visible columns when you insert a row and specify a _state of UNDEFINED. Ingres issues a runtime warning when this happens but does assign the column values. It is never possible to assign the DELETED state when inserting a row.

By default, if you do not assign a state to the row, the inserted row has a state of UNCHANGED. (For information about the table field row states, their meanings, and how they interact, see Table Fields (see page 579).)

Each column in the inserted row (except hidden columns) has its *change* variable cleared (set to 0).

If the table field contains any derived columns, the FRS calculates and displays the values for those columns, if possible, when a row is inserted.

When you execute a dynamic insertrow, the column names and values found in the specified *descriptor_name* are used. The *descriptor_name* identifies an SQL Descriptor Area (SQLDA), a host language structure allocated at run time. An SQLDA contains an array of elements called sqlvar elements. Each sqlvar contains fields to hold the data type, length, and name of a column in the table field. Each also points to a variable that holds the value for the column it describes. Before you can execute a dynamic insertrow statement, you must allocate the SQLDA, describe the table field, and allocate the necessary variables. See your query language reference guide and to your host language companion guide for complete instructions on these procedures. (The dynamic usage of insertrow is not available in QUEL.)

To turn display attributes on or off when the row is inserted, use the optional with *attribute* clause; specify 0 to turn the attribute off, 1 to turn the attribute on. In the case of color, specify a color code from 0 to 7. The attributes you specify are assigned to the value and scroll with the value. To assign attributes to an entire row, explicitly specify all column names. *Attribute* must be one of the values listed in the following table:

blink

Specifies that the field blinks on and off

color

Specifies that the field is displayed in the specified color (0 - 7)

intensity

Specifies that the field is displayed in half or bright intensity, depending on terminal

normal

Specifies no special attributes

reverse

Specifies that the field is displayed in reverse video

underline

Specifies that the field is underlined

Examples—insertrow statement:

Example 1:

Insert a new row as the first row in the table field display.

```
exec frs activate menuitem 'TopRow';
 exec frs begin;
    exec frs insertrow empform employee 0
      (ename = :ename, sal = :sal);
exec frs end;
```

Example 2:

Allow the user to insert a blank row into the table field before or after the current row.

```
exec frs activate menuitem 'InsertBefore';
 exec frs begin;
    exec frs inquire_frs table empform
      (:row = rowno(employee));
    row = row - 1;
    exec frs insertrow empform employee :row;
exec frs end;
 exec frs activate menuitem 'InsertAfter';
exec frs begin;
    exec frs insertrow empform employee;
exec frs end;
```

Example 3:

Provide a cut and paste facility, using the **deleterow** and **insertrow** statements.

```
exec frs activate menuitem 'Cut';
 exec frs begin;
    exec frs getrow (:ename = ename, :age = age);
    exec frs deleterow empform employee;
    cut = true;
exec frs end;

exec frs activate menuitem 'Paste';
exec frs begin;
    if (cut = false) then
        exec frs message
                    'You must select a row first';
        exec frs sleep 2;
    else
        exec frs insertrow empform employee
           (ename = :ename, age = :age);
        cut = false;
    end if;
exec frs end;
```

Example 4:

Insert a null into the spouse field and, using a *indicator_var*, provide the run time ability to insert a null into the title field.

```
exec frs activate menuitem 'NewEmployee';
 exec frs begin;
    exec frs insertrow empform employee
      (spouse = null, title = :title:indicator_var);
exec frs end;
```

Example 5:

Using dynamic statements, add a new row into the user's data set. The row is retrieved from the database. Also, assign the value 1 to the hidden column fromdb to indicate that this row is from the database.

```
exec frs activate menuitem 'InsertNext';
 exec frs begin;
    exec sql fetch csr using descriptor sqlda;
    /*
    ** Add an SQLVAR for the hidden column setting
    */
    sqlda.sqld = sqlda.sqld + 1;
    hide_col = sqlda.sqld;
    fromdbval = 1;
    sqlda.sqlvar(hide_col).sqltype = int;
    sqlda.sqlvar(hide_col).sqllent = 2;
    sqlda.slqvar(hide_col).sqldata =
                               address(fromdbval);
    sqlda.sqlvar(hide_col).sqlind = null;
    sqlda.sqlvar(hide_col).sqlname = 'fromdb';
    exec frs insertrow :form_var :table_var
                      using descriptor sqlda;
 exec frs end;
```

loadtable Statement—Append a Row of Data

This statement appends a row of data to a table field's data set.

Syntax

Non-dynamic version:

```
loadtable formname tablename
    (columnname = value {, columnname = value})
    [with (attribute(columnname) = 0 | 1 |color
    {, attribute(columnname) = 0 | 1 | color})]
```

Dynamic version:

```
loadtable formname tablename
    using [descriptor] descriptor_name
```

Description

The loadtable statement loads values into a table field's data set. The dynamic version enables the program to do this using values determined at run time. The loadtable statement is typically executed before an initialize statement or inside the initialize loop, to load a form's table field with a set of database values retrieved using a select loop or a cursor.

You can specify *columnname* using a quoted or unquoted string or host string variable. The data type of the *value* must be compatible with the data type of the column. The *formname* must specify the name of the form in which the table field *tablename* is displayed. You can specify *formname* and *tablename* using quoted or unquoted strings or host string variables.

Each time the loadtable statement is executed, one row is appended to the data set. As rows are added, they are displayed until the table field display is full. After the table field display is full, newly-loaded values continue to be added to the end of the data set; the user can access these rows by scrolling to them.

The non-dynamic loadtable statement loads values into columns in the table field. The list of *columnnames* identifies the columns receiving values and the values they are to receive. You can specify any hidden, invisible, or displayed column in the table field; you cannot load values into derived columns. Values for any derived columns are automatically calculated and loaded, if possible, when the source columns are loaded.

To set the state of a row in table field, specify the constant _state for *columnname*. Only specified columns receive values. Any column not included receives a null if it is nullable or a default value (a blank for character columns or 0 for numeric columns).

If you want to load a null into a column, there are two ways to do so. First, you can specify the column's associated *value* as the key word null. This method loads a null into the column whenever the statement executes. The alternate, and more flexible method, uses an indicator variable. Using an indicator variable allows the user or program to decide at run time whether to place a null in the column.

An indicator variable is a two-byte integer variable associated with the variable used to assign values into the column. To specify an indicator variable, use the following syntax:

```
loadtable formname tablename (columnname = var:indicator_var)
```

You can only use an indicator variable when you use a variable for *var*. Also, you must have previously declared the indicator variable in a host variable declaration section.

When an indicator variable is set to -1, any value in its associated variable is ignored and a null is assigned to the specified column. You must set the indicator variable before executing the loadtable statement and the receiving column must be nullable. See your host language companion guide for a complete description of the use of indicator variables.

If the *columnname* is the constant _state, *value* must evaluate to one of the following:

0 — UNDEFINED
1 — NEW
2 — UNCHANGED
3 — CHANGED

You must not assign values to any visible columns when you load a row and specify a _state of UNDEFINED. Ingres issues a runtime warning when this happens but does load the column values. It is never possible to assign the DELETED state when loading a row.

By default, if you do not explicitly assign a row state to a loaded row, the new row has the state of UNCHANGED. This state changes to CHANGED as soon as the user or the program alters any of the row's values. For a description of each row state, their meanings and interactions, see Table Fields (see page 579).

Each column in the loaded row (except hidden columns) has its change variable cleared (set to 0).

The loadtable statement can only be performed on a table field associated with a data set. Therefore, it must follow an inittable statement for the table field.

The dynamic version of the statement loads the table using column names and values found in *descriptor_name*. The *descriptor_name* identifies an SQL Descriptor Area (SQLDA), a host language structure allocated at run time. An SQLDA contains an array of elements called sqlvar elements. Each sqlvar contains fields to hold the data type, length, and name of a column in the table field. Each also points to a variable that holds the value for the column it describes. Before you execute a dynamic loadtable you must allocate the SQLDA, describe the table field, and allocate and set the necessary variables. See the *SQL Reference Guide* and your host language companion guide for instructions on these procedures. (The dynamic version of loadtable is not available in QUEL.)

To turn display attributes on or off when the row is loaded, use the optional with *attribute* clause; specify 0 to turn the attribute off, 1 to turn the attribute on; if you are assigning color, you must specify a color code in the range 0 to 7. The attribute(s) you specify are assigned to the value, and scroll with the value.

Attribute must be one of the values listed in the following table:

blink

Specifies that the field blinks on and off

color

Specifies that the field is displayed in the specified color (0 - 7)

intensity

Specifies that the field is displayed in half or bright intensity, depending on terminal

normal

Specifies no special attributes

reverse

Specifies that the field is displayed in reverse video

underline

Specifies that the field is underlined

Examples—loadtable statement:

Example 1:

Load data from variables into the ename and eno columns of the employee table field.

```
exec frs loadtable empform employee
    (ename = :ename, eno = :eno);
```

Example 2:

Append a row of constant values to the end of the data set.

```
exec frs loadtable empform employee
    (ename = 'johnson', eno = 25);
```

Example 3:

Load all values from the employee database table into the employee table field.

```
exec sql declare empcursor cursor for
    select eno, ename, age, sal, dept
    from employee;
 exec sql open cursor for read only;
 exec sql whenever not found goto done;
 loop until no more rows
    exec sql fetch empcursor into
        :eno, :ename, :age, :sal, :dept;
    exec frs loadtable empform employee
        (eno = :eno, ename = :ename, age = :age,
    sal = :sal, dept = :dept);
end loop;
 done:
     exec sql close empcursor;
```

Example 4:

Load information about an employee (gathered from another form) into a table field.

```
exec frs loadtable empform employee
    (ename = :name, spouse = :spouse:indicator_var,
    manager = null);
```

Example 5:

Using dynamic statements, retrieve data from a database and load it into a table field.

```
exec sql describe stmt into sqlda;
...
 exec sql declare csr cursor for stmt;
exec sql open csr for readonly;
 loop until no more rows
    exec sql fetch csr using descriptor sqlda;
    exec frs loadtable :form_var :table_var using
                descriptor sqlda;
end loop;
 exec sql close csr;
```

message Statement—Print a Message

This statement prints a message on the screen.

Syntax

```
message string
    [with style=menuline | popup
    [(option=value {, option=value})]]
```

Description

The message statement with no style clause or with style=menuline prints a one-line character string on the menu line. *String* must be either a quoted string literal or a program string variable. The maximum message length is the width of the screen. If the message length is greater than the width of the screen, only the characters that fit are displayed. If the message contains newline characters, the newlines are displayed as line breaks (carriage return-line feed).

A message statement that specifies with style=popup displays the message string in a pop-up box on the screen at a location specified by the programmer. The message can be more than one line. The message remains on the screen until the user enters a carriage return. (The FRS prompts for the carriage return following the message text.)

The starting location and size of the pop-up are determined by the *values* selected for the *options*. The *options* are:

startrow

Specifies the row position of the upper left hand corner of the pop-up

columns

Specifies the number of columns, including the borders, occupied by the pop-up

rows

Specifies the number of rows, including the borders, occupied by the pop-up

You can specify the values assigned to the options using integer literals, integer variables or default. If you specify default, the following FRS default values are used:

- **startcolumn**—The second column
- **startrow**—The size of the message determines the row position
- **columns**—The width of the screen minus 2
- **rows**—The number of rows required to display the message, plus one row for the carriage return prompt. The default row value cannot exceed the number of rows in the screen minus one row for the menu line.

The integer equivalent to the keyword default is 0.

The starting location, described by the startcolumn and startrow options, cannot be less than 1,1, which describes the screen origin at the upper left hand corner of the screen. Column and row values increase as you move to the right and down the screen, respectively.

If the starting location is defaulted, the FRS selects a location that displays the pop-up directly above the menu line.

The columns and rows options describe the size of the pop-up box. The box must be at least 16 columns by 4 rows, and must allow room for the borders of the box. If the box size and starting location force the box to be wholly or partially off-screen, the FRS attempts to adjust the starting location so that the entire box can be displayed on the screen. The message is not displayed unless the entire box fits on the screen.

It is good practice to follow a message statement that does not have a style=popup clause with a sleep statement, to ensure that the message remains visible long enough for the user to read it. However, if the statement following message is a statement that can require some time to process, do not add more time with a sleep statement.

Examples—message statement:

Example 1:

Display a message for three seconds.

```
exec frs message 'Please enter an employee number';
 exec frs sleep 3;
```

Example 2:

Display a message before initializing two forms. Do not sleep, as the forminit can take a few seconds to complete.

```
exec frs message 'Initializing forms ...';
 exec frs forminit empform, deptform;
```

Example 3:

Display a pop-up error message from a buffer using defaults for its position and size.

```
exec frs message :errorbuf with style=popup;
```

printscreen Statement—Copy the Displayed Screen

This statement prints or stores in a file a snapshot of what is currently displayed on the terminal screen.

Syntax

```
printscreen [(file = filename)]
```

Description

The printscreen statement prints or stores in a file a copy of the current form and the character representation of its associated data.

Filename identifies where the copy is sent. You can use a quoted or unquoted string literal or a program variable for *filename*.

To store the copy in a file, *filename* must specify a valid file name. If the file is an existing file, the screen snapshot is appended to the end of the file.

To send the copy to the printer, *filename* must be the word printer. The copy is sent directly to your installation's default printer.

If you do not include the printscreen argument, the copy is stored in the location specified by II_PRINTSCREEN_FILE (if II_PRINTSCREEN_FILE is defined). If II_PRINTSCREEN_FILE has not been defined, you are prompted for the name of the file. You can specify a file or printer. (See the *System Administrator Guide* for information about setting II_PRINTSCREEN_FILE.)

Examples—printscreen statement:

Example 1:

Send a copy of the form to the printer.

```
exec frs printscreen (file = 'printer');
```

Example 2:

Store a copy of the current form in the file designated by the filevar program variable, which is initialized at the start of the program.

```
exec frs prompt ('Specify default file for screens: ',
                 :filevar);
if (filevar = '' then
    filevar = 'printer';
end if;
...
exec frs printscreen (file = :filevar);
```

prompt Statement—Prompt the User for Input

This statement prompts the user for input.

Syntax

```
prompt [noecho] (string, string_var)
    [with style=menuline |
    popup [(option=value {, option=value})]]
```

Description

The prompt statement displays a prompt on the screen and accepts user input. *String*, which contains the text of the prompt, must be a quoted string literal or a program variable. *String_var* is a string variable to hold the user's response to the prompt. A maximum of 200 characters are returned; if the screen width is less than 200 characters, then the maximum number of characters returned corresponds to the screen width. If the user fails to respond to the prompt, and a timeout period is specified in the application, then a zero-length string is returned in *string_var* when timeout occurs.

If you omit the style clause or specify style = menuline, the prompt appears on the last line of the screen. In such cases, the maximum allowable length of the prompt is equal to the width of the screen. The user's response occupies an entire line for pop-up style prompts (unless you specify noecho).

When you specify the pop-up style, the prompt appears inside a pop-up box. Using the pop-up style allows you to display a multi-line prompt.

The *options* specify the location and size of the pop-up. Valid options are:

startcolumn

Specifies the column position of the upper left hand corner of the pop-up

startrow

Specifies the row position of the upper left hand corner of the pop-up

columns

Specifies the number of columns, including borders, that the pop-up occupies

rows

Specifies the number of rows, including borthat the pop-up occupies

An option *value* can be an integer literal, an integer variable, or the word default. Default values are the FRS default values:

- **startcolumn**—The second column
- **startrow**—Determined by the size of the *string*
- **columns**—The width of the screen minus 2
- **rows**—The number of rows needed to display the prompt, plus one row for your response. This value never exceeds the number of rows on the screen minus one row for the menu line.

The integer equivalent of the keyword default is 0.

The starting location, defined by startcolumn and startrow, cannot be less than 1,1. This pair of coordinates, located in the upper left hand corner of the screen, describes the origin point on the screen. Column and row values increase as you move to the right and down the screen, respectively.

If the starting location is defaulted, FRS selects a location that displays the prompt just above the menu line.

The columns and rows options determine the size of the pop-up box. The size cannot be smaller than 16 columns by 4 rows and must allow room for the border. If the box size and starting location, in combination, force the box to be wholly or partially off the screen, FRS attempts to adjust the starting location so that the entire box can be displayed. The prompt is not displayed unless the entire box is on the screen.

If the key word noecho is specified, the user's input is not displayed on the screen. Noecho is useful in situations where a password is required.

Examples—prompt statement:

Example 1:

Prompt the user for an employee's department.

```
exec frs prompt ('Enter the department: ', :deptvar);
```

Example 2:

Prompt for a password before opening a database cursor to view employee information.

```
exec frs prompt noecho ('Enter Password: ', :passwd);
 if (passwd = 'leviticus') then
    exec sql open viewemp;
else
    exec frs message 'No permission for task';
    exec frs sleep 2;
end if;
```

Example 3:

Use a prompt as an interactive message displayed at the top of the screen.

```
exec frs prompt ('An error has occurred
    [Press Return]', :var)
    with style=popup (startcolumn=1, startrow=1);
```

purgetable Statement—Purge Deleted Rows

This statement throws away the list of deleted rows from a table field's data set.

Syntax

```
purgetable [formname] [tablename]
```

Description

The purgetable statement removes any deleted rows from the deleted list of a table field's data set so that the memory held by these rows can be reused. The purgetable statement can only be used on table fields that have been initialized with the inittable statement. If there are no rows in the deleted list when a purgetable statement is issued no operation occurs, but time is wasted checking the list, and so unnecessary purgetable statements must be avoided.

Formname must identify a declared form (see the addform or forminit statement descriptions). You can specify *formname* and *tablename* using a quoted or unquoted character string literal or program variable. The *formname* must be the name of the form containing the specified table field. If formname evaluates to an empty string then the current form is assumed. Similarly, if *tablename* evaluates to an empty string then the current field is assumed to be the table field to be cleared. If the current field is not a table field then a runtime error results.

Examples—purgetable statement:

Example 1:

Clear the rows marked for deletion in the data set of the table field employee on the form empform.

```
exec frs purgetable empform employee
```

Example 2:

Clear the rows marked for deletion in the data set of the table field departments on the current form.

```
exec frs purgetable ' ' departments
```

Example 3:

Clear the rows marked for deletion in the data set of the table field contained in the program variable whichtable, on the current form.

```
exec frs purgetable ' ':whichtable
```

putform Statement—Transfer Data to the Form

This statement transfers data into the form.

Syntax

Non-dynamic version:

```
putform [formname]
    (fieldname = value{, fieldname = value})
```

Dynamic version:

```
putform [formname]
    using [descriptor] descriptor_name
```

Description

The putform statement transfers data into simple fields on a form. The dynamic version allows you to do this using values determined at run time.

Formname must identify a declared form (see the addform or forminit statement descriptions). You can specify *formname* using a quoted or unquoted character string literal or program variable. If you use putform in a display block, omit *formname*: the current form is assumed.

In the non-dynamic version, the list of *fieldnames* identifies the fields into which values are placed. The fields must be simple fields which are not derived fields. (Use the loadtable, insertrow, and putrow statements to transfer data into table field columns.) You can specify express *fieldname* using a quoted or unquoted character string or a program variable. Similarly, *value* can be either a literal or a variable. A field and the value assigned to it must have compatible data types. (See your host language companion guide for information about compatible data types.)

There are two ways to assign a null value to a field: (1) specify the field's associated *value* as the key word null, or (2) use an indicator variable. Using an indicator variable enables the user or program to decide at run time whether to place a null in the field. See your query language reference guide for descriptions of Indicator variables.

When you execute a dynamic putform, the field names and values found in the specified *descriptor_name* are used. The *descriptor_name* identifies an SQL Descriptor Area (SQLDA), a host language structure allocated at run time. See your query language reference guide and your host language companion guide for information about the SQLDA. (The dynamic usage of putform is not available with QUEL.)

Each field that receives new values has its change variable set to 0.

A validation check is *not* automatically performed when a value is assigned to a field. To check that the value assigned is valid according to the current specification for the form, use validate Statement—Validate Fields (see page 753).

Examples—putform statement:

Example 1:

Place data from a constant and a program variable into the form empform.

```
exec frs putform empform (ename = 'bill',
                          sal = :salvar);
```

Example 2:

Place data from the database into the current form.

```
...
 exec frs activate menuitem 'GetNext';
 exec frs begin;
    exec sql fetch cursor1 into :namevar, :salvar;
    exec frs putform (ename = :namevar,
                      sal = :salvar);
exec frs end;
```

Example 3:

Place a null into a field using the null constant.

```
exec frs activate menuitem 'NewOrder';
 exec frs begin;
     . . .
     exec frs putform (description = null);
     . . .
 exec frs end;
```

Example 4:

Use an indicator variable to provide the run time potential to put a null into a field.

```
exec frs activate menuitem 'Invoice';
 exec frs begin;
    . . .

    /* Display salesperson for invoice if any */
    exec frs putform (salesperson =
                               :name:indicator_var);
    . . .
 exec frs end;
```

Example 5:

Using dynamic statements, retrieve data from a database and display it on a form for the user to browse. (The cursor and the putform statement use the same SQLDA.)

```
exec frs describe form :form_var into sqlda;
...
 exec sql declare csr cursor for stmt;
exec sql open csr for readonly;
 loop until no more rows
    exec sql fetch csr using descriptor sqlda;
    exec frs putform :form_var using
        descriptor sqlda;
    exec frs sleep 4;
end loop;
 exec sql close csr;
```

putrow Statement—Update a Table Row

This statement updates values in a table field row.

Syntax

Non-dynamic version:

```
putrow formname tablename [row]
    (columnname = value {, columnname = value})
```

Dynamic version:

```
putrow formname tablename [row]
    using [descriptor] descriptor_name
```

Description

The putrow statement updates the values in a table field row. It does not add a new row to the table field.

The *formname* is the name of the form in which the table field *tablename* is displayed. Both can be specified using character strings, with or without quotes, or as program variables.

The *row* is the number of the displayed row into which the data are put. *Row* can be specified as an integer literal or an integer program variable, and must indicate a row currently displayed on the table field. For instance, if the table field can display four rows, but currently contains only two, the row number must be either 1 or 2. To update the current row, omit *row*.

If you execute this statement inside an unloadtable loop, you must omit *row*; putrow always updates the row currently being unloaded. (Any statement that changes values in the data set must be used with caution inside an unloadtable loop.)

In the non-dynamic version, the list of *columnnames* identifies the columns being updated and their new values. You can specify any hidden, invisible, or displayed column in the table field except derived columns. You can use a quoted or unquoted character string literal or a program variable for *columnname*. Only those columns that are specified receive values. The new values overwrite any values currently in the specified columns. Values in columns which are not specified in the argument list are not changed.

The *value* must be either a literal or a program variable and must have a data type compatible with the data type of its associated column.

To place a null in the column, you can (1) specify the column's associated *value* as the key word null, or (2) use an indicator variable. Using an indicator variable allows the user or program to decide at run time whether to place a null in the column. Indicator variables are described in your query language reference guide.

When you execute a dynamic putrow, the column names and values found in the specified *descriptor_name* are used. The *descriptor_name* identifies an SQL Descriptor Area (SQLDA), a host language structure allocated at run time. An SQLDA contains an array of elements called sqlvar elements. Each sqlvar contains fields to hold the data type, length, and name of a column in the table field. Each also points to a variable that holds the value for the column it describes. Before you can execute a dynamic putrow statement, you must allocate the SQLDA, describe the table field, and allocate the necessary variables. See the *SQL Reference Guide* and to your host language companion guide for complete instructions on these procedures. (The dynamic version of putrow is not available in QUEL.)

The putrow statement sets the change variable for each updated column to 0.

When a putrow statement is executed on a row in the data set, it can affect the row's state. If the row is in a state of UNCHANGED, putrow alters its state to CHANGED. However, if the row is in a state of NEW, putrow has no effect on the row's state. For a detailed description of row states, see Table Fields (see page 579).

The putrow statement can be used to change the state of a row in the data set. The valid values that _state can be set to depends on the current value of _state. If _state is NEW, then it can be set to NEW, UNCHANGED or CHANGED. If _state is UNCHANGED then it can be set to CHANGED. If _state is CHANGED then it can be set to UNCHANGED. The syntax for using the putrow statement to set a row's state is the same as the syntax above, simply use _state as the *columnname* and the value of NEW, CHANGED or UNCHANGED as the *value*. You can specify _state as an integer literal or an integer variable.

For example, to change the current row's state to UNCHANGED:

```
exec frs putrow empform employee (_state = 2);
```

Examples—putrow statement:

Example 1:

The PutCurrent operation places data in the table field row on which the cursor currently sits.

```
exec frs activate menuitem 'PutCurrent';
 exec frs begin;
    /* Put new information into the current row. */
    exec frs putrow empform employee
        (age = 52, sal = :sal);
```

Example 2:

The PutFirst operation puts data in the first displayed row of the table field.

```
exec frs activate menuitem 'PutFirst';
 exec frs begin;
    exec frs putrow empform employee 1
        (sal = :sal);
exec frs end;
```

Example 3:

As the employee table field is unloaded, use the putrow statement to mark the rows processed, so as not to process them again. Assume a hidden column marked specified with format integer1.

```
exec frs activate menuitem 'ProcessRecords';
 exec frs begin;
    exec frs unloadtable empform employee
        (:ename = ename, :age = age,
         :marked = marked, :state = _state);
    exec frs begin;
        /* Process if new, unchanged or changed */
        if ((state = 1 or state = 2 or state = 3)
            and (marked = 0)) then
            process the data;
            exec frs putrow empform employee
                (marked = 1);
        end if;
    exec frs end;
exec frs end;
```

Example 4:

Update a table field row, using an indicator variable to allow for potential nulls in the specified column. Menu item UpdateOrder takes information about an order from simple fields and updates a table field row with the changed data.

```
exec frs activate menuitem 'UpdateOrder';
 exec frs begin;
    /* Get data from other fields */
    exec frs getform sales (:name:indicator_var =
                             salesperson);
    /* Update table field row */
    exec frs putrow sales saleslog
        (ordernumber = :order, salesperson =
        :name:indicator_var);
exec frs end;
```

Example 5:

Place a null into a field using the null constant.

```
exec frs activate menuitem 'NewOrder';
 exec frs begin;
    . . .
    exec frs putrow sales saleslist
        (salesperson = null, ordernum = :order);
 exec frs end;
```

Example 6:
Put values in a table field using a descriptor.

```
exec frs putrow :formname
            :tablename using descriptor sqlda;
```

Example 7:

Process the records in the data set within a database multi-statement transaction. Reset the _state of any CHANGED or NEW rows so that a user can make further changes to the data set and they are correctly processed when this menu item is chosen again. Error handling is ignored.

```
exec frs activate menuitem 'Save';
 exec frs begin;
    exec sql savepoint startupdate;
    exec frs unloadtable empform employee
        (:ename = ename, :age = age, :eno = eno,
           :state = _state);
    exec frs begin;
        if (state = 0) then
            /* undefined is left alone */
            null;
        else if (state = 1) then
            /* new is appended */
            exec sql insert into employee
                (eno, ename, age)
                values (:eno, :ename, :age);

            /* reset _state to UNCHANGED */
            exec frs putrow empform employee
                (_state = 2);
        else if (state = 2) then
            /* unchanged is left alone */
            null;
        else if (state = 3) then
            /* Reflect changed data */
            exec sql update employee
                set ename = :ename, age = :age
                where eno = :eno;
            /* reset _state to UNCHANGED */
            exec frs putrow empform employee
                (_state = 2);

        else if (state = 4) then /* deleted row */
            exec sql delete from employee
                where eno = :eno;
        end if;
    exec frs end;
    exec sql commit;
exec frs end;
```

redisplay Statement—Refresh the Screen

This statement refreshes the screen and displays the current values of fields.

Syntax

```
redisplay
```

Description

The redisplay statement updates the screen with the current field values for all the forms that are currently displayed. Ordinarily, changes made to data in a form's fields by the program during an operation are not actually displayed on the form until all statements in the operation have been executed. The redisplay statement provides the capability to refresh the screen and display the current values at an earlier point. For example, you can use redisplay to refresh the screen immediately following a putform statement.

The statement must be within the scope of a display loop. However, it does not have to be syntactically located in the actual display block; it can be in a called routine.

The forms system avoids unnecessary refreshes of the screen. If it thinks that a refresh produces a screen that reproduces exactly what is currently on display, then it does not refresh the screen after a redisplay or resume statement. There are cases where the forms system must refresh the screen, even though it thinks a refresh is not needed; for example, when changes are made to the screen using host language write statements, because the FRS does not know about those changes.

You can force the forms system to refresh the screen by issuing a clear screen statement prior to the redisplay. Thus the following commands cause the screen to be cleared and the current form to be displayed again:

```
exec frs clear screen;
 exec frs redisplay;
```

This looks exactly like what happens when the user enters the FRS command redisplay (Ctrl-W on many terminals).

Example—redisplay statement:

Put data in the key field and refresh the screen so that the data is visible. Then, retrieve the values from a complicated query that can take some time to process.

```
exec frs activate menuitem 'Key';
 exec frs begin;
    exec frs clear field all;
    exec frs putform (key = :keyvar);
    exec frs redisplay;
    call exec_query(keyvar);
exec frs end;
```

resume Statement—Resume the Display Loop

This statement positions the cursor and resumes the display loop.

Syntax

```
resume
resume next
resume field fieldname
resume column tablename columnname
resume menu
resume entry
resume nextfield
resume previousfield
```

Description

The resume statement terminates the currently executing operation and returns control to the FRS. When control resumes, the screen cursor is situated at one of several possible locations, depending on the syntax of the resume.

If an operation does not have a resume statement, once the operation is completed the cursor resumes on the same field as before the operation. Thus, the default action at the end of an operation is equivalent to a simple resume statement, without any object names.

The resume statement must be syntactically within an activate or initialize block, as it generates a local goto statement to the beginning of the display block. If the resume statement is located within an unloadtable statement nested within an activate section, resume exits the unloadtable loop, in addition to terminating the activate operation and resuming the display loop.

If a resume statement positions the cursor to a field or column that has entry activations, the activations are executed.

Syntax of Resume	**Screen Cursor Location**
resume	Cursor is positioned on same field or table field column as before the operation.
resume next	Cursor is positioned to reflect the cursor movement that initiated a field or column activation.
resume field *fieldname*	Cursor is positioned on *fieldname.*
resume column *tablename columnname*	Cursor is positioned on *columnname.*
resume menu	Cursor is positioned on the menu line.
resume entry	Cursor is positioned on the current field or table field column and any entry activation for that field or column is performed.
resume nextfield	Cursor is positioned on the first accessible field or table field column that is next in the tab sequence.
resume previousfield	Cursor is positioned on the first accessible field or table field column that is before the current field in the tab sequence.

The Resume Next Statement

The resume next statement must appear within an activate field or activate column section. Resume next directs the FRS to continue with whatever operation the user selected that caused the activation.

Resume next behaves differently depending on whether the activation is the result of a cursor movement out of a field or a menu key, FRS key, or menu item selection:

- Activation initiated by cursor movement

  Resume next causes completion of the cursor movement. For example, if an attempt to move forward out of a field initiated a field activation operation, resume next positions the cursor on the next field in the form. Similarly, if the field activation had been initiated by an attempt to move the cursor to the previous field, the resume next positions the cursor on the previous field in the form. In the case of column activation, resume causes the cursor to resume on the next column, the previous column, the row above or the row below, depending on what action precipitated the activation. If a scroll attempt had initiated the activation, a scroll occurs.

  For example, assume that resume next is included in the activation block for the field emp_name and that the program turns on previousfield activations:

  ```
  exec frs set_frs frs (activate(previousfield) = 1) ;
  ```

  Then, when the user types the FRS previousfield command (Ctrl P on many terminals) while the cursor rests on the field emp_name, the field's activate block runs. The resume next in the activate block moves the cursor to the previous field.

- Activation as the result of a menu key, FRS key, or menu item selection.

  The selection is suspended while the field activation is executing. To complete the suspended selection, use resume next in the activation block; other resume statements causes the suspended selection to be discarded.

  Selections are completed as follows:

  – Menu key selection

    The cursor is placed on the menu line, ready for menu item input.

  – FRS key selection

    The block of code associated with the selected FRS key is executed.

  – Menuitem selection

    The block of code associated with the selected menu item is executed.

The Resume Field Statement

In the resume field variant, *fieldname* can be specified using a string, with or without quotes, or as a string program variable. The field can be either a simple field or a table field. If the field is a table field, the cursor is placed on the first row and first column, unless the cursor was in the table field when the resume field statement executed. In such cases, the cursor resumes on the current row.

The Resume Nextfield Statement

The resume nextfield variant puts the cursor on the first accessible field or table field column that is next in the tab sequence. If the current field is a table field column, then the next field is the first accessible column to the right in the table field. If the current field is the last accessible column in a table field, the cursor is moved off the table field to the first accessible field, or table field column, that is after the table field in the tab sequence.

If there are no other accessible fields on the form then no operation occurs if a resume nextfield statement is issued.

The Resume Previousfield Statement

The resume previousfield variant puts the cursor on the first accessible field or table field column that is before the current field in the tab sequence. If the current field is a table field column, then the previous field is the first accessible column to the left in the table field. If the cursor is already on the first accessible column in a table field, then the cursor is moved off the table field to the first accessible field, or table field column, that is before the table field in the tab sequence.

If there are no other accessible fields on the form then no operation occurs if a resume previousfield statement is issued.

Other Resume Statements

In the resume column variant, *tablename* and *columnname* can be expressed as strings, with or without quotes, or as program variables.

You cannot use the resume menu and resume entry variants to resume on a regular submenu.

Examples—resume statement:

Example 1:

Terminate the operation, with cursor on field empname.

```
exec frs resume field empname;
```

Example 2:

Terminate the operation, with cursor on the menu line.

```
exec frs resume menu;
```

Example 3:

Terminate a field or column activation operation, and complete the cursor movement that initiated the operation, or complete a suspended menu key, FRS key, or menu item selection.

```
exec frs resume next;
```

Example 4:

Provide a menu item to re-initialize the current field by executing the field entry block again. When the resume entry statement executes, the field activation block is executed.

```
exec frs activate menuitem 'Reinitialize Field';
 exec frs begin;
    exec frs message 'Reinitializing field';
    exec frs sleep 2;
    exec frs resume entry;
exec frs end;
 exec frs activate before field 'fld1'
exec frs begin;
    exec frs set_frs field '' (reverse(fld1)=1);
exec frs end;
```

scroll Statement—Perform a Table Field Scroll

This statement performs a table field scroll.

Syntax

```
scroll formname tablename to record | end

scroll formname tablename up | down
```

Description

The scroll statement executes a table field scroll. You can use this statement to scroll to a particular record in the data set, to the end of the data set, or simply to scroll up or down one record. Scrolling to a particular record (or the end) is known as a *target scroll*.

The *formname* identifies the form in which the table field *tablename* is displayed. Use quoted or unquoted character string literals or program variables to specify these parameters.

When you perform a target scroll using the to version of the syntax, you must specify either the end of the data set or a particular record in the data set. You can determine the record number of any row in the data set using the _record constant; see the getrow and unloadtable descriptions for more information about this. *Record*, which can be either a integer literal or integer variable, must evaluate to a positive integer. Consequently, deleted rows, which have negative record numbers, cannot be the target of a scroll.

The row scrolled to becomes the current row of the table field. Therefore, if a resume field statement is then issued for the table field, the cursor is positioned on that row.

Specifying up or down performs the same type of scrolling as do FRS commands. When used for a table field with a data set, it simply moves the data set rows up or down one line in the display. For instance, the scroll up version scrolls the top row of the table field out of sight. All other records in the table field's display are moved up one row, and the next record in the data set (if any) appears as the last row in the display. Because the FRS already provides this functionality this type of scroll is rarely used.

When a scroll occurs, not only do the values for each column scroll, the change variable associated with each value also scrolls, even into the data set if necessary.

Examples—scroll statement:

Example 1:

These operations perform target scrolls to the bottom, top, and middle of the data set.

```
exec frs activate menuitem 'Bottom';
 exec frs begin;
   exec frs scroll empform employee to end;
exec frs end;
 exec frs activate menuitem 'Top';
exec frs begin;
    exec frs scroll empform employee to 1;
exec frs end;
 exec frs activate menuitem 'Middle';
exec frs begin;
    inquire_frs table empform
        (:rows = datarows(employee));
    rows = rows/2;
    if (rows > 0) then
       exec frs scroll empform employee to :rows;
    end if;
exec frs end;
```

Example 2:

Find a particular record in the data set.

```
exec frs activate menuitem 'Find';
 exec frs begin;
    /* Prompt for name to search for */
    exec frs prompt ('Lastname to search for: ',
               :searchname);
    /*
    ** Loop through data set and stop when name is
    ** found.
    */
    exec frs unloadtable empform employee
        (:ename = ename, :record = _record);
    exec frs begin;
        if (ename = searchname) then
            /* Scroll to record with specified name*/
            exec frs scroll empform employee
                to :record;
            exec frs resume field employee;
        end if;
    exec frs end;

    /*
    ** All rows in data set searched and name not
    ** found.
    */
    exec frs message 'Cannot find named employee.';
    exec frs sleep 2;
exec frs end;
```

set_frs Statement—Set FRS Features

This statement sets a variety of FRS features.

Syntax

```
set_frs object_type {parent_name}[row_number]
    (frs_constant[(object_name)] = value
    {, frs_constant[(object_name)] = value})
```

Description

The set_frs statement provides a way to dynamically set various features of the FRS (FRS). Among these are display attributes for fields and table field columns, activation of field validations, timeout periods, and mappings for control, function, and arrow keys.

The *object_type* specifies the type of object for which features are being set. It must be expressed as a string, with or without quotes. The following are valid object types for the set_frs statement:

column

Sets attributres of a column in a table field

form

Sets attributres of a form

field

Sets attributres of a field in a form

frs

Sets attributres of the FRS

menu

Sets attributres of the menu for the current display loop

row

Sets attributres of a row in a table field

The *parent_name* identifies the form or table field that contains the specific object affected by the *frs_constant*. You can use a quoted or unquoted character string literal or a program variable to express the *parent_names*. The specific object is identified by the *object_name*. All *object_names* referenced in a single set_frs statement must belong to the same set of parents. That is, all specified fields must be on the same form and all columns must be part of the same table field. The parent name(s), combined with the object name, allow an *frs_constant* to reference a specific object. If a parent name is left as a quoted empty or blank string, the current parent is used.

The frs and form object types do not require any parent names.

The following table lists the valid *parent_names* for each *object_type*:

Object Type	**Parent Name**
column	formname tablename
field	formname
form	None
frs	None
menu	*formname* (Optional; always refers to the current display loop)
row	formname tablename

The *frs_constant* is the name of one of the FRS constants. These constants identify the type of feature being set. These are the valid constants:

activate

Turns field activation mechanisms on or off

active

Enables or disables a specified menu item

change

Sets the change variable of a form, field, or column

color

Sets color in terminals supporting color

display

Turns display attributes on or off. *Display* can have a value of reverse, blink, underline, intensity, normal, invisible, or displayonly.

editor

Enables or disables the FRS editor command

format

Changes the format of simple field or column of a table field

getmessages

Enables or suppresses error messages on field value retrieval

inputmasking

Enables or disables usage of a FRS format template for a field, column or form

label

Defines an alias for a control, function, or arrow key mapped to a menu item, FRS key, or FRS command

map

Sets mappings between control/function/arrow keys and FRS keys, FRS commands, and menu items

mapfile

Specifies the application's mapping file

menumap

Turns display of menu item/function key associations on or off

mode

Sets the display mode for a form or tablefield

outofdatamessage

Enables or disables the Out of Data message displayed by the FRS when a user attempts to scroll off the bottom or top of a table field

rename

Renames a specified menu item

shell

Enables or disables the FRS shell command

timeout

Establishes a timeout period of specified length for an application

validate

Turns field validation mechanisms on or off

Each of these constants is described in its own section following.

The *object_name* identifies the actual object to which each FRS constant refers. Use a quoted or unquoted string literal. If the *object_name* refers to a form object, you can also use a program variable. The object name can vary from FRS constant to FRS constant within a statement. When object type is frs, the object is an *FRS object*; otherwise, it is a form object. If object name is not specified where normally a *form* object is required, the current form object is used.

The object names for applicable object types are listed in the following table:

Constant	Object
column	columnname
field	fieldname
form	formname
frs	*frs_object* (varies, according to FRS constant)
row	columnname

The *value* must be a value permitted by its FRS constant, as described in the individual sections following.

A corresponding inquire_frs statement exists for each set_frs statement. Thus, dynamic querying and setting of features can be performed from within an application program.

Setting Global Field and Column Validations

The validate constant is used to turn the various mechanisms for field and table field column validation checks on and off. (The validation checks themselves are specified for the fields and columns at the time the form is defined in VIFRED.) It sets these validations for the application as a whole, rather than for merely a single form object; therefore, it requires the frs object type. The syntax for this particular variant of the set_frs statement is:

```
set_frs frs
    (validate(validation_name) = value
    {, validate(validation_name) = value})
```

The *validation_name* describes the type of validation mechanism being set. *Value* must be either 1 or 0, or the name of an integer variable with the value 1 or 0. The value 1 turns the validation mechanism on; 0 turns it off. The *validation_names*, their default *values* and the type of validation mechanism that each controls are listed in the following table:

Validation Name	Default Value	Type of Validation
keys	0 (off)	Pressing any control/function/arrow key mapped to an FRS key.
menu	0 (off)	Pressing the Menu Key.
menuitem	0 (off)	Selecting any menu item (or function/ control/arrow key) mapped to a menu item.

Validation Name	Default Value	Type of Validation
nextfield	1 (on)	Moving to the next field or column.
previousfield	0 (off)	Moving to the previous field or column.

These validations have a global effect. For example, the statement

```
exec frs set_frs frs (validate(keys) = 1);
```

causes a validation to be performed for the current field whenever *any* control, function, or arrow key mapped to an FRS key is pressed. Field validations, once set, apply to every form in the application (that is, they are global to the application).

The following statement turns on all possible field and column validations for every form in the application:

```
exec frs set_frs frs ( validate(nextfield) = 1,
                        validate(previousfield) = 1,
                        validate(menu) = 1,
                        validate(keys) = 1,
                        validate(menuitem) = 1 ) ;
```

The above is often issued right after the exec frs forms statement.

Setting Global Field and Column Activations

The activate constant controls field and column activations for the application as a whole rather than individually specified fields and columns. You can use it to override the default behavior of the activate conditions. By default, if you define an entry or exit field activation, that activation occurs only when the user enters or exits, respectively, the specified field. If you want the field activation to occur when the user performs some other activity, such as pressing the Menu key, you can use the appropriate set_frs statement to obtain that result.

For example, assume that an exit activation is defined for the emp_name field. In addition, the program has previously executed the following set_frs statement:

```
exec frs set_frs frs (activate(menu)=1)
```

After issuing the preceding set_frs statement, whenever the user presses the Menu key while the cursor rests on the emp_name field, the FRS performs the field activation before performing any operation associated with the Menu key. (In addition, if the field activation does not include a resume next, the operation associated with the Menu key is never performed.)

The syntax for this particular variant of the set_frs statement is:

```
set_frs frs
     (activate(activation_type) = value
     {, activate(activation_type) = value})
```

The *activation_type* specifies the condition for an activation to occur. *value* must be 0, 1, or the name of an integer variable with the value 0 or 1. The value 1 turns on the activation condition; 0 turns it off. The *activation_type* names, their default values and the condition that each controls are listed in the following table; these values are the same as for the validation names:

Activation Name	Default Value	Activation Condition
before	1 (on)	Entering a field or column
keys	0 (off)	Pressing any control, arrow, or function key mapped to an FRS key
menu	0 (off)	Pressing the Menu Key
menuitem	0 (off)	Selecting any menu item (or function, arrow, or control key mapped to a menu item)
nextfield	1 (on)	Moving to the next field or column
previousfield	0 (off)	Moving to the previous field or column

Field activations, once set, apply to every form in the application (that is, they are global to the application). For example, the following statement turns on two possible field and column activations for every form in the application and is often issued immediately after the forms statement:

```
exec frs set_frs frs (activate(nextfield) = 1,
 activate(previousfield) = 1);
```

As another example, the following statement causes an activation to occur on the current field whenever any menu item is selected by the runtime user:

```
exec frs set_frs frs (activate(menuitem) = 1);
```

The activate column statement *always* executes when the user moves up or down a line in the table field. This activation is not settable. Setting the logical/environment variable II_FRS_ACTIVATE to 60 automatically sets all global field and column validation/activation options to 1 at the time the forms statement is executed.

Setting Field and Column Display Attributes

You can assign display attributes to:

- Individual fields
- Table field columns
- Individual values in table field data sets
- Individual cells in table fields The FRS combines the various attributes assigned to a cell. For color, the lowest color code is used.

Assign Display Attributes to Individual Fields

To assign display attributes to fields:

```
set_frs field formname
     (display[(fieldname)] = value
     {, display[(fieldname)] = value})
```

Assign Display Attributes to Table Field Columns

To assign display attributes to table field columns:

```
set_frs column formname tablename
    (display[(columnname)] = value
    {, display[(columnname)] = value})
```

Assign Display Attributes to Table Field Data Set Values

You can assign display attributes to individual data set values or cells in a table field. When you assign a display attribute to a value, the attribute scrolls with the value; when you assign a display attribute to a cell, the attribute does not scroll.

To assign display attributes to a value in a table field data set, you must use the set_frs row statement within an unloadtable loop. You must omit row number. The syntax is:

```
set_frs row formname tablename
     (display(columnname) = value
    {, display(columnname) = value})
```

Explicitly name the display attributes in the statement for each column you change. To apply a change to all columns in a table field you must have a (*display*(*columnname*) = *value* for each column.

You can also assign display attributes to table field values using the insertrow and loadtable statements; see the command descriptions in this chapter for more information.

Assign Display Attributes to Table Field Cells

To assign display attributes to a specific cell in a table field, you must specify row number as well as column name. The syntax is:

```
set_frs row formname tablename rowno
     (display(columnname) = value
    {, display(columnname) = value})
```

Display Parameters

You can specify form objects using (quoted or unquoted) string literals or host string variables.

Value must be 0, 1, or an integer host variable having a value of 0 or 1. 1 turns the specified display attribute on; 0 turns the attribute off. If you are assigning a color, *value* must specify the color code, from 0 to 7.

Valid values for the *display* parameter are:

blink

Specifies that the field blinks on and off

color

Specifies that the field is displayed in the specified color (0 - 7)

displayonly

Specifies that the field is displayed but the user cannot change it; invalid for table fields and set frs row

intensity

Specifies that the field is displayed in half or bright intensity, depending on terminal

invisible

Specifies that the field is not displayed

normal

Specifies no special attributes

reverse

Specifies that the field is displayed in reverse video

underline

Specifies that the field is underlined

Each display attribute for a field is by default turned off, unless it was turned on when the form was defined in VIFRED or by a previous set_frs statement. The set_frs statement overrides any previous setting of a particular display attribute for a field, and can be used to turn off any attributes previously turned on. For details about display attributes, see the *Character-based Querying and Reporting Tools User Guide.*

The normal display attribute actually signifies the lack of any display attribute. Setting normal to 1 causes any blink, intensity, reverse, and underline display attributes currently in effect for a field, column, table field value or cell, to be turned off. Setting normal to 0 has no effect.

The FRS displays the changes when the application program exits the code block that contains the set_frs statement. To display the change before the end of the code block, follow the set_frs statement with a redisplay statement.

Examples—set_frs statement:

Example 1:

The following statement causes the current field in the current form to appear in reverse video and blink:

```
exec frs set_frs field ' ' (reverse = 1, blink = 1)
```

Example 2:

The following statement turns off blinking in the partname column of the table field partstbl in the parts form:

```
exec frs set_frs column parts partstbl
    (blink(partname) = 0)
```

Setting Field and Column Color

The color of fields and table field columns can be set dynamically by means of the **color** constant. The syntax for setting the color of fields is:

```
set_frs field formname
    (color[(fieldname)] = value
    {, color[(fieldname)] = value})
```

The syntax for setting the color of table field columns is similar:

```
set_frs column formname tablename
    (color[(columnname)] = value
    {, color[(columnname)] = value})
```

The form objects can be expressed as string constants, with or without quotes, or as program string variables.

Value must be either an integer in the range of 0 to 7, or the name of an integer variable with a value in that range. The *value* indicates the color code for the field or column. The color codes can be associated with the color settings for a terminal that supports color by means of the terminal's termcap entry. (See *Character-based Querying and Reporting Tools User Guide* for details on creating termcap entries.) Ingres comes with predefined termcap entries for the following terminal types that support color, listed in the following table:

Terminal Type	**Designation**
Envision	envisionc
DEC VT241	vt241
Tektronix 4105	tk4105c

The default color code is 0, indicating a terminal's default foreground color.

Setting the Change Variable

There is a change variable associated with each field in a form and each displayed column in a table field. If a table field has a data set, then each value in the data set has an associated change variable. The change variable for a field is set if a user types into the field. The clearrest FRS command, mapped to the Return key on many terminals, sets the change variable. The change variable is cleared at the start of a display loop, when the field is cleared, or when a new value is placed into the field by the program.

To set the change variable for a field, use the following syntax:

```
set_frs field formname
    (change[(fieldname)] = value
    {, change[(fieldname)] = value})
```

To set the change variable for a table field, use the following syntax:

```
set_frs row formname tablename [row]
    (change(columnname) = value
    {, change(columnname) = value})
```

The form objects can be expressed as string constants, with or without quotes, or as program string variables.

The *row* identifies a row in the referenced table field display. If the table field has a data set, then *row* can refer only to rows that actually have displayed data. For example, if the table field can display five rows, but only three actually display data when the statement is issued, then *row* can be 1, 2, or 3. The first row in the table field is always numbered 1. If you do not specify *row*, the statement refers to the row on which the cursor is currently resting.

You can never specify *row* if the statement appears in an unloadtable loop. In such cases, the statement always refers to the row just unloaded.

Value must be either an integer literal of 0 or 1 or an integer variable evaluating to 0 or 1. A value of 0 clears the change variable to make it appear that the runtime user had not typed into the field. The value 1 sets the change variable.

The following statement clears the change variable for a field:

```
exec frs set_frs field empform (change(empname) = 0);
```

The following statement sets the change variable for two columns in a table field:

```
exec frs set_frs row partsform partstable 1
    (change(partnumber) = 1, change(partdescription) = 1);
```

There are additional examples at the end of the inquire_frs statement description.

Setting Control, Function, and Arrow Key Mappings

The map constant provides the means to dynamically map menu items, FRS keys, and FRS commands to function, control, and arrow keys during the running of the application. Key mapping at the application level takes precedence over installation and user mapping files. Function and control key mapping is discussed in detail in the *Character-based Querying and Reporting Tools User Guide.*

The syntax for this particular variant of the set_frs statement is:

```
set_frs frs
    (map(menuN|frskeyN|frs_command) = pfN|controlX/arrow_key
    {, map(menuN|frskeyN|frs_command) =
    pfN|controlX/arrow_key})
```

where *arrow_key* can be any of:

uparrow
downarrow
leftarrow
rightarrow

The object name for this statement can be menu*N*, frskey*N* or *frs_command*. The menu*N* object designates the *N*th menu item on the menu line. *N* must be in the range of 1 to 25. *Frs_commands* are built-in FRS functions that can be mapped to control, function, or arrow keys. They are described in detail in the *Character-based Querying and Reporting Tools User Guide.* The frskey*N* object represents any of the FRS keys in use during the application. Here, *N* must be in the range of 1 to 40.

Any of the objects described above can be mapped to a function, control, or arrow key, which must be specified on the right of the equal sign. A function key is designated by pf*N*, where *N* is in the range of 1 to 40. Control*X* designates a control key. *X* can be any *single* letter of the alphabet, or the designations del (indicating the delete key) or esc (indicating the escape key). Arrow keys are specified using the key words listed above.

For example, the statement

```
exec frs set_frs frs (map(frskey5) = pf3);
```

equates FRS key 5 with function key 3. Therefore, the block of code associated with an activate frskey statement utilizing FRS key 5 is activated when the user selects function key 3.

Similarly, the statement

```
exec frs set_frs frs (map(menu3) = controlh);
```

allows the code activated by the third item on the menu line to be alternatively activated by pressing the control key and the letter H on the keyboard.

You can map each FRS key, FRS command, and menu item to only a single control, function, or arrow key. Conversely, each control, function, and arrow key can be mapped to only one FRS key, FRS command, or menu item at each display level. This means that a key that is mapped to an FRS key, FRS command, or menu item for the top level display block can be mapped differently for a submenu within that display block.

Defining an Alias for Mapped Keys

The label constant defines an alias for a key that is mapped to a menu item, FRS key, or FRS command. Because not all terminal keyboards are the same, using an alias can make it easier for the runtime user of an application to quickly find the correct control or function key. By specifying the letters on a keyboard key cap as an alias in a label statement, a runtime user need not memorize the layout of a keyboard or go through a trial and error period. If defined, aliases are displayed for mapped menu items on the menu line and for all mapped objects in the help facility.

The syntax for the label variant of the set_frs statement is:

```
set_frs frs
    (label(menuN|frskeyN|frs_command) = value
    {, label(menuN|frskeyN|frs_command) = value})
```

Value defines the alias. You can use a string literal or a program string variable for *value*. The label statement accepts the exact same set of object names as the map statement.

As an example, assume that the third function key on a keyboard has the letters CONT on it and that the following statement is executed in an application:

```
exec frs set_frs frs (map(menuitem1) = pf3);
```

The menu line looks like this:

```
Run(PF3)   Help   End
```

The quick way for a runtime user to select the first menu item is to press PF3. However, because there is no key with the letters PF3 on it, the user must mentally make the association between pf3 and the key that has CONT printed on it. This makes the application harder to use. However, if the application also contains the following statement:

```
exec frs set_frs frs (label(menuitem1) = 'CONT');
```

then the menu line is

```
Run(CONT)   Help   End
```

The user can easily see which key to press.

Any mappings or aliases created by the set_frs statement remain in effect for the remainder of the application unless overridden by another map or label statement.

Aliases can be specified in mapping files. See the *Character-based Querying and Reporting Tools User Guide* for a complete discussion on mapping files.

Setting the Menu Map

The menu map displays the association between the menu items and any control, function, or arrow keys to which they are mapped. The menumap constant turns the menu map on or off. The syntax for setting the menu map is:

```
set_frs frs (menumap = value)
```

The menumap constant does not take an *object_name*.

Value must be either 1 or 0, or an integer variable evaluating to 1 or 0. 1 turns the display of the menu map on; 0 turns the display off if it is currently on. The default setting for the menu map
is on.

The following example turns on the menu map and sets validation on the Menu key:

```
exec frs set_frs frs (menumap=1, validate(menu)=1);
```

Specifying the Mapping File

The mapfile constant allows you to specify an application's FRS key mapping file at run time. This is a quick way to execute many set_frs map and/or label statements. The syntax is:

```
set_frs frs (mapfile = filename)
```

The mapfile constant does not take an *object_name*.

The *filename* must be either a character string literal or a string variable. It is recommended that the complete specification be given for the file.

Further information on mapping files can be found in the *Character-based Querying and Reporting Tools User Guide.*

Setting the Display Mode of a Form

The display mode of a form can be set using the mode constant. The syntax is:

```
set_frs form (mode[(formname)] = modetype)
```

You can specify both *formname* and *modetype* as quoted or unquoted string literals or as program string variables. The *modetype* must evaluate to one of the following: fill, update, read, or query.

The following statement changes the mode of the current form to the read mode:

```
exec frs set_frs form (mode = read);
```

Note: If a form containing a table field is set to read mode, the table field mode is not changed; however, the table field responds as if it is in read mode.

Setting the Display Mode of a Table Field

You can set a table field's display mode at run time using the mode constant. The syntax for this is:

```
set_frs field formname (mode(fieldname) = value)
```

You can use either character string literals or program variables to represent the *fieldname*, *formname*, and *value*. The *value* must evaluate to one of the following: fill, update, read, or query.

If you change the table field's display mode to or from query, the data set is cleared. For a complete description of the effects on the table field when you change from one mode to another, see Table Fields (see page 579).

Setting a Form's Change Variable

Use the change constant to set a form's change variable. The syntax is:

```
set_frs form (change[(formname)] = value)
```

You can express the *formname* using a string literal, with or without quotes, or as a program string variable.

Value must be either 1 or 0, or an integer variable evaluating to 1 or 0. The value 1 indicates that a change has been made to the displayed data on the form; 0 indicates no change to the displayed data. This variant of the set_frs statement is particularly useful for resetting the change value to 0, because the value is automatically changed to 1 when the data is changed. A corresponding inquire_frs statement queries the current value of change.

Setting the Timeout Period

The timeout period is the length of time that the forms system waits for a user to take an action. If the user does not make a requested response or move the cursor within the specified time, the forms system returns control to the application. The timeout constant establishes the timeout period. The syntax for this is:

```
set_frs frs (timeout = value)
```

A timeout period can only be specified for user-written, forms-based applications.

Value specifies the length of the timeout period and can be an integer constant or integer variable. The timeout's length cannot be greater than 32767 seconds. A timeout value of 0 means that no timeout occurs. If the timeout value is less than 0, the value is forced to 0 and a runtime error is issued.

Note: For timeout periods less than 10 seconds, results are very system dependent. Some environments have problems working with small values for timers. Others experience problems with small values only when heavily loaded. For these reasons, setting your timeout period to less than 10 seconds can affect the portability of your application.

For a full discussion of the timeout feature, see Table Fields (see page 579). The activate command description provides a discussion of activate timeout blocks, which can be used in applications to handle a display loop that times out.

Enabling the FRS shell Command

The FRS shell command enables your application to spawn (create) a shell process.

You can use a set_frs statement to enable or disable this ability. (By default, this ability is turned off.) The syntax is:

```
set_frs frs (shell = 0|1|integer_variable)
```

To disable the shell command, specify 0. Setting shell to 1 enables the command. If you use an integer variable, it must evaluate to either 0 or 1.

See the *Character-based Querying and Reporting Tools User Guide* for a complete description of the FRS commands.

Enabling the FRS editor Command

The FRS editor command starts a text editor. By default, this command is turned on. You can use set_frs statement to disable or enable this feature; the syntax is as follows:

```
set_frs frs (editor = 0|1|integer_variable)
```

To disable the editor command, specify 0; to enable the editor command, specify 1. If you use an integer variable, it must evaluate to either 0 or 1.

See the *Character-based Querying and Reporting Tools User Guide* for a complete description of the FRS commands.

Suppressing Validation Error Messages

When an application retrieves values from fields on a form into variables, the FRS performs a variety of validation checks to ensure that the data is valid before assigning it into the variables. The FRS returns error messages if it encounters invalid data. You can use set_frs to suppress the display of validation error messages that might result from the following statements:

getform
getrow
unloadtable

The syntax is:

```
set_frs frs (getmessages = 0|1|integer_variable)
```

To disable the display of validation error messages, specify 0. To enable display of validation error messages, specify 1. By default, display is enabled. If you use an integer variable, it must contain 0 or 1.

How You Can Change a Field's Format

The format constant allows you to change the display format of a simple field or a column of a table field. The syntax for changing the format of a simple field is

```
set_frs field form_name (format(field_name) = value)
```

The syntax for changing the format of a column of a table field is:

```
set_frs column form_name table_name
    (format(columnname) = value)
```

The new format is subject to the following constraints:

- It must be compatible with the field's data type.
- The number of rows and columns cannot exceed the size of the original format.
- Any data in the field must be valid.

Shown below are examples of changing a field's format:

- To change a date field to display dates in a format of *ddmmmyyy* (for example, 14 Jan 1999):

  ```
  exec frs set_frs field myform
      (format(datefld)= 'd"03 Feb 1901"');
  ```

- To change a numeric field to display monetary values as $nnn,nnn,nnn.nn:

  ```
  exec frs set_frs field myform
      (format(salary)='"nnn,nnn,nnn.nn"');
  ```

- To change a floating point column to display three values after the decimal point:

  ```
  exec frs set_frs column myform mytable
       (format(floatcol)='f12.3');
  ```

See the *Character-based Querying and Reporting Tools User Guide* for a complete discussion of form field display formats.

Renaming Menu Items

To rename individual menu items for a display loop, use the rename option:

```
set_frs menu formname | empty_string
        (rename(old_name) = new_name)
```

To refer to the current form you can specify an empty string instead (*empty_string* must be "" in ESQL, ' ' in EQUEL). You can rename disabled menu items. If two menu items have the same name, the FRS operates on the first occurrence of the name.

Enabling and Disabling Menu Items

To enable or disable individual menu items for a display loop, use the active option:

```
set_frs menu formname | empty_string
     (active(menuitem = 0 | on | 1 | off | :int_var)
```

where 0 or **on** enables the specified menu item and 1 or **off** disables the menu item. You can use a host language integer variable to specify the setting (0 or 1). To refer to the current form you can specify an empty string instead (*empty_string* must be "" in ESQL, ' ' in EQUEL).

When you disable a menu item, it is removed from the menu line and cannot be selected by the user. The remaining menu items are redisplayed, and function key mappings are adjusted. For example, if the following menu line is displayed:

```
One(PF1)   Two(PF2)   Three(PF3)   Four(PF4)
```

and your application disables the menu item Two:

```
exec frs set_frs menu (active(Two) = off);
```

the resulting menu line looks like this:

```
One(PF1)   Three(PF2)   Four(PF3)
```

If you disable all menu items, the user must press the Menu key to exit the display loop.

Enabling and Disabling the Out of Data Message

By default, the FRS displays an Out of Data message when a user attempts to scroll off the top or bottom of a table field. To disable or re-enable this message, use the outofdatamessage option:

```
set_frs frs (outofdatamessage = 0 | off | 1 | on | 2 | bell |
    :int_var)
```

Valid settings are:

0 or off

Suppresses Out of Data message

1 or on

Displays Out of Data message

2 or bell

Suppresses Out of Data message; rings terminal bell once

sleep Statement—Suspend Application Execution

This statement suspends execution of the application for a specified number of seconds.

Syntax

```
sleep seconds
```

Description

The sleep statement suspends the execution of the application for a specified number of seconds. You must use an integer literal or variable to specify the *seconds*.

This statement is particularly helpful when used in conjunction with the message statement to ensure that the message remains on the screen long enough for the user to read it. (However, if the statement following message is a statement that can require some time to process, adding more time with a sleep statement is unnecessary.)

Examples—sleep statement:

Example 1:

Display a message for three seconds.

```
exec frs message 'Please enter an employee number';
 exec frs sleep 3;
```

Example 2:

If an expert mode flag is set, display messages for a shorter period of time.

```
if (expert_mode) then
    msg_interval = 1;
else
    msg_interval = 3;
end if;
    ...
exec frs message 'Entering information form';
exec frs sleep :msg_interval;
```

submenu Statement—Start a Menu Not Linked to a Form

This statement starts a menu not directly linked to a form.

Syntax

```
submenu
```

Description

The **submenu** statement displays a menu, known as a *submenu*, that is not directly linked to a displayed form. (Submenus linked with a form are displayed with the **display submenu** statement and described in Table Fields (see page 579).) You can use a submenu as a nested menu inside a display loop or as a multi-item prompt without an attached form. Submenus can provide a greater level of detail to the user.

When the user selects a menu item operation in which a submenu is nested, the submenu replaces the original menu line for the form. The user can then choose an operation from the submenu. When the chosen operation is finished, the original menu line appears; there is no implicit looping for submenus.

The submenu statement signals the start of a submenu. You can use only the activate menuitem, activate frskey, and activate timeout statements to create submenu operations. (An activate timeout section does not define a menu line operation.) The activation sections for the operations must directly follow the submenu statement. The submenu ends with the last consecutive activate section. The submenu statement and its associated activate sections are called a submenu block.

Do not place any other statements or any comment lines between the submenu statement and the first activate section or between the activate sections themselves. The FRS assumes that any such statements or comments indicate the end of the submenu.

You cannot issue the resume menu or resume entry statement in a submenu block, nor use it to resume on a submenu. (Use the display submenu statement to resume on the submenu.)

Example—submenu statement:

Provide a submenu for the Select operation.

```
exec frs activate menuitem 'Select';
 exec frs begin;
    exec frs submenu;
    exec frs activate menuitem 'Employee';
    exec frs begin;
        /* Select an employee record */
    exec frs end;
    exec frs activate menuitem 'Manager';
    exec frs begin;
        /* Select a manager record */
    exec frs end;
exec frs end;
```

tabledata Statement—Traverse Columns

This statement traverses the columns in a table field.

Syntax

```
tabledata
begin
    program code;
end
```

Description

The tabledata statement works in conjunction with the formdata statement to create a loop through all the columns in the current table field, executing the same section of code for each column. You must embed this statement inside a formdata statement.

The statement's begin/end block contains the code that are executed for each column in the table field. The inquire_frs statement is often part of this code, so that the program can make inquiries about each column in the table field. This is particularly useful if the program does not know which form is in use until run time. The template for this usage is:

```
formdata formname
begin
    ...

tabledata
    begin
            inquire_frs on current column;
            program code;
    end
    ...

end
```

The full range of inquire_frs column variants, as well as all other embedded SQL and host language statements, can appear in the tabledata loop. (See the inquire_frs statement description for a complete list of its possibilities.) The tabledata loop simply starts the first pass on the first column in the current table field and sequences along to the next column with each additional pass through the loop.

You can use the endloop statement to break out of the loop when necessary. See the endloop statement description (see page 646) for its syntax and usage.

Example—tabledata statement:

Loop through a form, printing out all field and column names.

```
exec frs formdata :formname;
 exec frs begin;
    exec frs inquire_frs field ''
        (:fldname = name, :istable = table);
    if (istable = 1) then
        print fldname,' is a table field';
        print '---------------';
        exec frs tabledata;
        exec frs begin;
            exec frs inquire_frs column '' ''
                (:colname = name);
            print colname, ' is a column';
        exec frs end;
        print '---------------';
    else
        print fldname, ' is a regular field';
    end if;
exec frs end;
```

unloadtable Statement—Loop Through Rows and Execute Statements

This statement loops through the rows in the data set and executes statements using values from those rows.

Syntax

Non-dynamic version:

```
unloadtable formname tablename
    (variable:indicator_var = columnname
    {, variable:indicator_var = columnname})
begin
    program code;
end
```

Dynamic version:

```
unloadtable formname tablename
    using [descriptor] descriptor_name
begin
    program code;
end
```

Description

The unloadtable statement enables a program to access values in each row of the data set in turn and perform actions based on those values. When a table is unloaded, the program takes values from the first row of the data set, places them into specified program variables, and executes statements using those values. It then unloads the second row and executes statements based on values in that row. The loop continues through all the rows in the data set. The actions performed on each set of values are defined by the code inside the statement's begin/end block. (If there is no code in the block, the statement merely scans the rows, performing no actions.)

The dynamic version of this statement places the values into variables pointed to and described by *descriptor_name*. *Descriptor_name* identifies an SQL Descriptor Area, which is a host language structure allocated at run time. For information about the structure, allocation, and use of the SQLDA, see the *SQL Reference Guide* and your host language companion guide. (The dynamic version of unloadtable is not available in QUEL.)

You cannot place any other statements or comment lines between the unloadtable statement and it associated begin/end block.

The *formname* is the name of the form in which the table field *tablename* is displayed. *Tablename* must identify an initialized table field (a table field that has an associated data set). You can use a quoted or unquoted character string literal or a string variable to *specify formname* and *tablename*.

The *columnname*s specify the columns to be unloaded into variables. (The program code in the begin/end block can only access values from columns that are explicitly named; values in columns that are not unloaded cannot be accessed by the code.) The column can be any hidden or displayed column or the special constants, _state and _record. The _state constant returns an integer corresponding to the row's state; for a complete discussion of row states and their meanings and interactions, see Table Fields (see page 579). The _record constant returns an integer record number corresponding to the row currently being unloaded, with 1 signifying the first row in the data set. If the row is a deleted row, its record number is negative. The data types of a column and its associated *variable* must be compatible. (See your query language reference guide for information about data types and your host language companion guide for information about compatible data types.)

If a specified column is nullable, you must use an indicator variable in conjunction with the column's *variable*. Ingres returns an error if it attempts to unload a null and no indicator variable is specified. Indicator variables are described in the your query language reference guide.

The unloadtable statement can access the values of deleted rows; however, when you delete undefined or new rows, they are removed from the data set, and cannot be accessed.

You can include the validate statement in an unloadtable loop, to verify the data being unloaded. If any of the data fails the validation, the unloadtable loop is aborted. You can also terminate the loop with the endloop or resume statements, described in this chapter.

Avoid using the following statements in the begin/end block of the unloadtable statement: loadtable, deleterow, insertrow, inittable, and clear field *tablefield_name*. Because these change the order or number of rows in the data set, using them in an unloadtable loop can have unforeseen consequences.

Because the *tablename* must identify an initialized table field, the unloadtable statement must follow an inittable statement for the table field. A common location for unloadtable is at the end of the form's display loop, so that the values in the data set can be processed after all changes have been made to them. An unloadtable statement cannot be nested within another unloadtable statement.

To set or inquire about change variables associated with values in a table field data set that is not currently displayed, you must use the inquire_frs row or set_frs row statement, described earlier, within an unloadtable block. Within an unloadtable block, you must omit the row number; these statements always refer to the row just unloaded.

Examples—unloadtable statement:

Example 1:

Process the records in the data set within a database multi-statement transaction. Error handling is ignored.

```
exec sql savepoint startupdate;
 exec frs unloadtable empform employee
    (:ename = ename, :age = age, :eno = eno,
           :state = _state);
exec frs begin;
    if (state = 0) then
        /* undefined is left alone */
        null;
    else if (state = 1) then /* new is appended */
        exec sql insert into employee
                  (eno, ename, age)
           values (:eno, :ename, :age);

    else if (state = 2) then
        /* unchanged is left alone */
        null;
    else if (state = 3) then
        /* Reflect changed data */
        exec sql update employee
            set ename = :ename, age = :age
            where eno = :eno;
    else if (state = 4) then
        /* deleted row */
        exec sql delete from employee
            where eno = :eno;
    end if;
exec frs end;
exec sql commit;
exec frs clear field employee;
```

Example 2:

In the form statistics with a table field accumulator containing columns name and number, add the integer values of number and put the average in a field called avgvalue. Stop on first error. Assume all the rows are in the state NEW.

```
rows = 0;
 sumvals = 0;
 exec frs unloadtable statistics accumulator
  (:name = name, :number = number, :record = _record);
exec frs begin;
    if (name = ' ') then
        exec frs message 'Empty numeric identifier';
        exec frs sleep 2;
        exec frs scroll statistics accumulator
                          to :record;
        exec frs resume field accumulator;
                          /* Break out of loop */
    end if;

    rows = rows + 1;
     sumvals = sumvals + number;
exec frs end;
 if (rows > 0) then
   sumvals = sumvals/rows;
end if;
 exec frs putform statistics (avgvalue = :sumvals);
```

Example 3:

Find the first employee over the age of 50.

```
found = false;
 exec frs unloadtable empform employee
    (:ename = ename, :age = age, :state = _state,
     :record = _record);
exec frs begin;
    if (age > 50) and (state \ 4) then
       /* Not deleted */
        found = true;
        exec frs endloop;
    end if;
exec frs end;
 if (found) then
    process the specified record;
end if;
```

Example 4:

Unload the values from a table field, using an indicator variable to allow for a potential null in its one nullable column.

```
exec frs activate menuitem 'PrintRoster';
 exec frs begin;
    exec frs unloadtable empform employee
        (:name = empname, :age = age,
         :spouse:indicator_var = spouse);
     exec frs begin;
         /* Do processing to create roster entry. */
    exec frs end;
exec frs end;
```

Example 5:

Check the change variable of rows in a table field data set.

```
exec frs activate menuitem 'PrintOrders';
 exec frs begin;
    exec frs unloadtable salesform saleslist
        (:ordernum = order, :item = item);
    exec frs begin;
        /* Check if item ordered has changed. */
        exec frs inquire_frs row salesform saleslist
            (:changed = change(item));
        /* If item that is ordered has changed, */
        /* update the database. */
        if (changed = 1) then
            update the database
        end if;
        process order and print out data;
     exec frs end;
exec frs end:
```

Example 6:

Using dynamic statements, unload a table field and insert the results into a database table using the same SQLDA.

```
exec frs describe table :form_var :table_var
    into sqlda;
build and prepare 'insert_stmt' from description of
    table field;
...
 exec frs unloadtable :form_var :table_var using
                             descriptor sqlda;
exec frs begin;
    exec sql execute insert_stmt using
        descriptor sqlda;
exec frs end;
```

validate Statement—Validate Fields

This statement performs the field validation checks specified in VIFRED.

Syntax

```
validate [field fieldname]
```

Description

The validate statement performs validation checks on a specified field or on all fields on a form. (For a description of validation checks, see Field Validation in Table Fields (see page 579).) If one or more fields fail a check, Ingres displays an error message, the application breaks out of the operation currently in progress, and the cursor is positioned on the first field that failed the check. If all the indicated fields pass the validation check, execution continues with the next statement.

When you execute this statement, the FRS performs the validation check immediately, independent of any user action. In contrast, the set_frs statement directs the FRS to perform the check when the user performs some specified action, such as pressing the Menu key.

If you omit the field *fieldname* clause, the FRS validates all fields and datatypes on the current form that have a validation check specified in VIFRED. The cursor is returned to the first invalid field, if any.

If you do specify a particular field, only that field is checked. The *fieldname* can identify a simple field or a table field. For table fields, the FRS validates the data in all of the *displayed* rows. (Data in unseen rows is automatically validated when the rows are scrolled out of the display.) The FRS does not validate data in hidden columns.

To validate only a single displayed row, use the validrow statement.

You can specify *fieldname* as a character string, with or without quotes, or as a program variable.

The validate statement must be syntactically within a display block, as it generates a local goto statement to the beginning of the block.

Examples—validate statement:

Example 1:

Validate the data in the field sal before getting the value.

```
exec frs validate field sal;
 exec frs getform empform (:salvar = sal);
```

Example 2:

Validate the data in the field indicated by the program variable fieldvar.

```
exec frs validate field :fieldvar;
```

Example 3:

Validate the displayed rows of the employee table field before unloading its data set.

```
exec frs validate field employee;
 exec frs unloadtable empform employee
    (:ename = ename, :age = age, :eno = eno);
exec frs begin;
    process the row;
exec frs end;
```

validrow Statement—Check Column Validity

This statement performs a validation check on columns in a specified row of the table field.

Syntax

```
validrow formname tablename [row]
    [(columnname {, columnname})]
```

Description

The validrow statement validates the columns in a specified displayed row of the table field. (For a full description of validation checks, see Table Fields (see page 579).) If a column fails a check, Ingres displays an error message, the application breaks out of the operation currently in progress, and the cursor is positioned on the first invalid column.

If all indicated columns pass the validation check, execution continues with the next statement. Only a row's displayed columns can be validated. (Hidden columns are not validated.) The validrow statement must be used within a display loop.

When you execute the validrow statement, the FRS performs the validation check immediately, independent of any user action. This is in contrast to the **s**et_frs statement, which directs the FRS to perform the check when the user performs some specified action, such as pressing the Menu key.

The *formname* is the name of the form in which the table field *tablename* is displayed. You can specify *formname* and *tablename* using quoted or unquoted character string literals or program variables.

The *row* is the number of the displayed row to be validated. *Row* can be either an integer literal or an integer program variable. The *row* must be in the range from 1 to the number of displayed rows for the table field as specified in VIFRED. If *row* is omitted, the row on which the screen cursor currently rests is validated.

The *columnnames* identify the columns to be validated in the specified row. You can specify a *columnname* as a character string, with or without quotes, or as a program variable. Only the columns listed are validated. If you omit the column list, the FRS validates all columns.

If you want to validate all displayed rows in a table field, use the validate statement.

Examples—validrow statement:

Example 1:

Validate all displayed columns of the current row, before issuing a getrow statement.

```
exec frs validrow empform employee;
 exec frs getrow empform employee
    (:ename = ename, :age = age, :eno = eno);
```

Example 2:

Validate the sal and age columns of the first displayed row before retrieving the values. Neither of these columns is hidden.

```
exec frs validrow empform employee 1 (age, sal);
 exec frs getrow empform employee 1
    (:age = age, :sal = sal, :state = _state);
```

Appendix I: EQUEL/FORMS Examples

This section contains the following topics:

This appendix contains examples of the EQUEL/FORMS statements, similar to the SQL examples provided in Forms Statements (see page 613). It also contains extended examples similar to those in Embedded Forms Program Structure (see page 541) and Table Fields (see page 579).

Activate

Declares a section of program code within a display block; the code is executed when specified conditions occurred.

Examples—activate statement:

Example 1:

The following example creates a help menu item:

```
## activate menuitem "help"
##  {
##    help_frs (subject = "application",
##             file = "application.hlp")
##  }
```

Example 2:

In the following example, the code block executes when the user leaves the field specified by the variable fieldvar. If the code block detects no errors, it issues a resume next statement to advance the cursor to the next field. If an error is detected, the cursor does not advance.

```
##  activate field fieldvar
##  {
     if (no error) then
##        resume next
     end if
##  }
```

Example 3:

In the following example, the FRS validates the current field when the user selects the **Next** menu item or presses the function, control, or arrow key mapped to FRS key 3. The code in the block is executed only if the current field passes the validation.

```
## activate menuitem "next" (validate=1),
## frskey3 (validate=1)
## {
##   retrieve cursor cursor1 (vname, vage)
##   putform empform (ename = vname, age = vage)
## }
```

Example 4:

In the following example, the application displays help text if the user types a question mark (?) and moves the cursor out of the key field; otherwise, the users entry is validated.

```
##  activate field key
##  {
##   getform keyform (vkey = key)
     if (vkey = '?') then
##        help_frs (subject = "keys", file = "keys.hlp")
##        resume
    end if
    found = 0
##  retrieve (found = 1) where keys.key = vkey
    if (found = 1) then
##       resume next
    else
##       message "unknown key, please modify."
##       sleep 2
      end if
##    /* default action is to resume on same field */
##  }
```

Example 5:

This example disables the scrolldown operation for the employee table field.

```
## activate scrolldown employee
## {
## }
```

Example 6:

This example initiates a database validation check on the ename column before scrolling to the next row in the employee table field.

```
##  activate scrollup employee
##  {
##    getrow empform employee (vname = ename)
##         retrieve (rowcount = count(employee.empname
##              where employee.empname = vname))
      if (rowcount = 0) then
##         message "the employee entered does not exist"
##         sleep 2
      else
##         scroll empform employee up
    end if
##  }
```

Example 7:

In the following example, when the user attempts to move the cursor out of the sal column, the activate column block is executed. If the value in sal is valid, the program continues with whatever action initiated the activate column block. If the user was attempting to move the cursor down a row while on the last displayed row of the table field, the activate scrollup block is executed at this time. If, however, the value in sal is not valid, the cursor resumes on the original column and row, and the activate scrollup statement is not executed.

```
## activate column employee sal
## {
     if (sal < 40000.00) then
##        resume next
     end if
##   message "reduce salary"
##   sleep 2
##   /* by default, the screen cursor resumes on the same
##   ** column. */
## }
## activate scrollup employee
## {
     program validation code
##   scroll empform employee up
## }
```

Example 8:

This example shows how the activate clause can be used to simplify data checking. The user cannot exit the field emp_name unless emp_name contains a valid name.

```
## activate field emp_name
## {
     check that emp_name contains a value
     if fail then
##        resume
     end if
##   resume next
## }
## activate menuitem "run" (activate = 1)
## {
     do processing based on value in field emp_name
## }
## activate frskey3 (activate = 1)
## {
     give raise to employee named in field emp_name
## }
```

Addform

Declares a compiled form to the FRS.

Examples—addform statement:

Example 1:

The following example declares the form empform:

```
## empform external integer
## addform empform
```

Example 2:

In the following example, a flag is used to insure that the addform statement is not issued more than once for this form:

```
##  empform external integer
##  added integer /* statically initialized to 0 */
    if (added = 0) then
##       addform empform
         added = 1
    end if
```

Example 3:

The following example illustrates use of the pound sign (#) to de-reference references to empform:

```
##  empform external integer
##  addform empform
##  display #empform
##  activate menuitem "end"
##  {
##       getform #empform (vname = ename)
##  }
```

Breakdisplay

Terminates a display loop without performing field validation or executing the **finalize** statement.

Examples—breakdisplay statement:

Example 1:

In the following example, the Quit operation terminates display of the form without validation checking:

```
## display empform
## initialize
## activate menuitem "Browse"
## {
     browse and update the data on the form
## }
## activate menuitem "Quit"
## {
##   breakdisplay
## }
## finalize
```

Example 2:

In the following example, a table field is unloaded within the Scan menu section. If an error is detected, breakdisplay terminates both the unloadtable code block and the display block.

```
## display empform
## activate menuitem "Scan"
## {
##   unloadtable empform employee (vchild = child,
                                vage = age)
##   {
          if (error) then
##             message "Aborting scan on error"
##             sleep 2
##             breakdisplay
          end if
##   }
## }
## finalize
## /* breakdisplay transfers control here. */
```

Example 3:

The following example illustrates the use of the breakdisplay statement in a display submenu display block:

```
## display "Form"
## activate menuitem "utilities"
## {
##   display submenu
##   activate menuitem "Delete"
##   {
          do delete based on data on form
##   }
##   activate menuitem "file"
##   {
          place data on form into a file
##   }
##   activate menuitem "End"
##   {
          /* exit from the submenu display block */
##        breakdisplay
##   }
##   finalize
## }
## activate menuitem "Done"
## {
##   /* exit from form display block */
##   breakdisplay
## }
## finalize
```

Clear

Clears the screen, individual fields on the form, or all fields on the form.

Examples—clear statement:

Example 1:

The following example clears the screen and displays a message:

```
## clear screen
## message "initializing ..."
```

Example 2:

The following example clears the data in the salary and dept fields:

```
## clear field salary, dept
```

Example 3:

The following example clears the data in the field specified by the program variable fieldvar:

```
## clear field fieldvar
```

Example 4:

The following example deletes the data set from the employee table field:

```
## clear field employee
```

Clearrow

Clears data from a row in a table field.

Examples—clearrow statement:

Example 1:

The following example clears all values in the current row:

```
## clearrow empform employee
```

Example 2:

The following example clears the ename column in the first displayed row:

```
## clearrow empform employee 1 (ename)
```

Deleterow

Deletes a row from a table field.

Examples—deleterow statement:

Example 1:

In the following example, the Delete menu item deletes the current row and saves the employee number for use in updating the database; if the row was a NEW row, appended by the run-time user, the row is not used to update the database.

```
## activate menuitem "delete"
## {
##   deleterow empform employee out (vnum = eno,
##                                  state = _state)
     if (state = 2 or state = 3) then
##       /*
##       ** the state indicates that the row was loaded
##       ** by program from the database. (it may have
##       ** since been modified.)
##       */
##       delete employee where employee.eno = vnum
     end if
## }
```

Example 2:

This example deletes the current row and retrieves the values of the deleted row into result variables using a param out target list:

```
   addresses(1) = address_of(vname)
   addresses(2) = address_of(vage)
   target_string = "%c = ename, %i4 = age"
## deleterow empform employee out
                        (param(target_string, addresses))
```

Display

Displays a form.

Examples—display statement:

Example 1:

The following example displays empform in update mode, with only an initialize and finalize statement:

```
## display empform update
## initialize (ename = vname, age = vage)
## finalize (vname = ename, vage = age)
```

Example 2:

The following example displays empform in the default fill mode, with two activate sections and no initialize or finalize statements:

```
## display empform
## activate menuitem "fill"
## {
      program code
## }
## activate menuitem "end"
## {
##    breakdisplay
## }
      program code
```

Example 3:

The following example displays empform in a pop-up window with no border, starting in the third row:

```
## display empform with style=popup
                        (border=none, startrow=3)
```

Display Submenu

Displays a submenu.

Example—display submenu statement:

The following example illustrates use of the display submenu to define a submenu that is displayed when the user selects the menu item option1:

```
## activate menuitem "option 1"
## {
##   display submenu
##   activate menuitem "option 1a"
##   {
##       program code
##   }
##   activate menuitem "quit"
##   {
##    enddisplay
##   }
## }
```

Enddisplay

Terminates a display loop.

Examples—enddisplay statement:

Example 1:

In this example, the End operation terminates display of the form:

```
## display empform
## initialize
## activate menuitem "browse"
## {
     browse and update the data on the form
## }
## activate menuitem "end"
## {
##   enddisplay
## }
## finalize (vname = ename, vage = age)
```

Example 2:

In this example, a table field is unloaded within the Scan menu activate section. If an error condition is detected, enddisplay terminates both the display and the unloadtable.

```
## display empform
## activate menuitem "scan"
## {
##   unloadtable empform employee
##        (vchild = child, vage = age)
##   {
          if (error) then
##             message "ending the scan"
##             sleep 2
##             enddisplay
          end if
##   }
## }
## finalize (vname = ename, vdept = dept, vsal = sal)
/* enddisplay transfers control to the finalize statement ** above.
*/
```

Example 3:

The following example uses the enddisplay statement in a display submenu display block:

```
## display empform
## activate menuitem "utilities"
## {
##   display submenu
##   activate menuitem "delete"
##   {
          do delete based on data on form
##   }
##   activate menuitem "file"
##   {
          place data on form into a file
##   }
##   activate menuitem "end"

##   {
           /* exit from the submenu display block */
           /* after validating data on form */
##         enddisplay
##   }
##   finalize (vbalance = balance)
## }
## activate menuitem "done"
## {
     /* exit from form display block */
     /* after validating data on form */
##   enddisplay
## }
## finalize (vtotal = total )
```

Endforms

Terminates an application program's connection with the FRS.

Examples—endforms statement:

Example 1:

This example severs the application's connection with the FRS:

```
## forms
   /*
   ** forms statements are allowed only within this scope
   */
## endforms
```

Example 2:

This example clears the screen before exiting from the FRS and the database:

```
## clear screen
## message "exiting application"
## endforms
## exit
```

Endloop

Terminates a loop.

Examples—endloop statement:

In this example, endloop is used to break out of an unloadtable loop on an error:

```
## unloadtable empform employee (vname = ename)
## {
     program code
     if (error) then
           /* break out of unloadtable statement loop */
##          endloop
     end if
## }
```

This example nests a display loop within an unloadtable statement. The endloop statement implicitly breaks out of the nested display loop.

```
## unloadtable formnames formtable
##    (vname = fname, vmode = fmode)
## {
##    display fname fmode
##    initialize
##    activate menuitem "next"
##    {
##         breakdisplay
##    }
##    activate menuitem "quit"
##    {
##         endloop
##    }
##    finalize
##    message "next form"
     /* breakdisplay transfers control here */
##    sleep 2
## }
     /* endloop transfers control here */
```

Finalize

Performs final data transfers at the end of a display loop.

Examples—finalize statement:

Example 1:

This example places data from the ename and sal fields into program variables:

```
## finalize (vname = ename, vsal = sal)
```

Example 2:

This example places data from the field specified by the variable fieldvar into the variable namevar:

```
## finalize (namevar = fieldvar)
```

Example 3:

The following example illustrates the use of a null indicator variable with a nullable column:

```
## finalize (spousevar:indicator_var = spouse)
```

Formdata

Loops through all fields in the form.

Example—formdata statement:

This example loops through a form, printing out all field and column names:

```
## formdata formname
## {
##   inquire_frs field "" (fldname = name,istable = table)
     if (istable = 1) then
         print fldname, " is a table field"
##       tabledata
##       {
##             inquire_frs column "" "" (colname = name)
##             print colname, " is a column"
##       }
    else
         print fldname, " is a simple field"
    end if
## }
```

Forminit

Declares an uncompiled form to the FRS.

Examples—forminit statement:

Example 1:

This example declares the uncompiled forms empform and deptform:

```
## forminit empform, deptform
```

Example 2:

This example insures that the forminit statement is not executed more than once for this form:

```
## added integer /* statically initialized to 0 */
   if (added = 0) then
##   message "initializing form..."
##   forminit empform
##   added = 1
   end if
```

Forms

Invokes the FRS for the application.

Example—forms statement:

This example invokes the FRS for a forms application:

```
## forms
     /*
     ** forms statements are allowed only within this
     ** block.
     */
## endforms
```

Getform

Transfers data from the form into program variables.

Examples—getform statement:

Example 1:

The following example places data from the ename and sal fields of form empform into program variables:

```
## getform empform (vname = ename, vsal = sal)
```

Example 2:

The following example places data from the field specified by the variable fieldvar, in the form specified by the variable formvar, into the variable namevar:

```
## getform formvar (namevar = fieldvar)
```

Example 3:

The following example places data from the current form into a database table. Within a display block, the form name need not be specified:

```
## activate menuitem "add"
## {
##   validate
##   getform (vname = ename, vsal = sal)
##   append to employee (ename = vname, sal = vsal)
## }
```

Example 4:

The following example illustrates the use of a null indicator variable when retrieving a nullable value from a field:

```
## getform empform (spousevar:indicator_var = spouse)
```

Example 5:

This example retrieves the current values from a form and puts them into the database using getform and append param target lists:

```
addresses(1) = address_of(vnum)
addresses(2) = address_of(vname)
addresses(3) = address_of(vage)
addresses(4) = address_of(vsal)
get_target_string = "%i4 = eno, %c = ename,
                      %i4 = age, %f4 = sal"
put_target_string = "eno = %i4, ename = %c,
                      age = %i4, sal = %f4"
## getform empform (param(get_target_string, addresses))
## append to employee (param(put_target_string,
##                        addresses))
```

Getoper

Gets comparison operators from fields and columns of a form displayed in query mode.

Example—getoper statement:

The following example builds a query using the getoper function. This example uses a query operator array that maps the integer codes of FRS query operators to their corresponding query language operators. Array subscripts begin at 1.

```
## oper_chars array(6) of character_string(2)
## voper      integer
## vnum       integer
## where_clause character_string(30)
   oper_chars = ("= ", "!=", "<", ">", <=", ">=")
## display empform query /* use query mode */
## activate menuitem "retrieve"
## {
##   getform empform (vnum = eno, voper = getoper(eno))
     if (voper > 0) then
##      /* construct where_clause like: e.eno = 18 */
        where_clause = "e.eno " + oper_chars(voper) + enum
     else
        /* nothing entered, use "truth" default */
        where_clause = "1 = 1"
     end if
##   retrieve (vnum = e.eno, vname = e.ename,
##              vage = e.age)
##          where where_clause
##    {
            process rows
##    }
## }
```

Getrow

Gets values from a table field row.

Examples—getrow statement:

Example 1:

This example gets information from the first row of the table field display:

```
## activate menuitem "getfirst"
## {
##    getrow empform employee 1
##         (vnum = eno, vname = ename, vage = age,
##          vsal = sal, vdept = dept)
## }
```

Example 2:

This example finds out if the current row has been modified:

```
## activate menuitem "rowchanged?"
## {
##    getrow empform employee
##         (vname = ename, state = _state)
      if (state = 1 or state = 3) then
      /* new or changed */
           msgbuf = "you have modified " + ename
##         message msgbuf
##         sleep 2
      end if
## }
```

Example 3:

This example illustrates the use of null indicator variables when getting a nullable value from a column in a table field:

```
## activate menuitem "moreinfo"
## {
##   getrow empform employee
##        (vname = ename, vage = age,
          vspouse:indicator_var = spouse)
     /* -1 means no spouse or children */
     if (indicator_var <> -1) then
          find information about children, if applicable
     end if
     display more detailed information
                  on retrieved employee
## }
```

Example 4:

This example retrieves the current values of row 2 and replaces that row in the database using getrow and replace param target lists.

```
addresses(1) = address_of(vage)
addresses(2) = address_of(vsal)
    get_target_string = "%i4 = age, %f4 = sal"
    put_target_string = "age = %i4, sal = %f4"
## getrow empform employee 2
## (param(get_target_string, addresses))
## replace employee (param(put_target_string, addresses))
```

Helpfile

Displays a file as help text.

Example—helpfile statement:

The following example shows the use of the helpfile statement to provide a help facility for the personnel form:

```
## activate menuitem "help"
## {
##   helpfile "personnel form" "personnel.hlp"
## }
```

Help_frs

Displays help.

Example—help_frs statement:

The following example provides help for the personnel form. By selecting the menu item Help, the user gains access to the Ingres help facility. The user can obtain information on the current form using the text contained in personnel.hlp. Using help, the user can look up field validations and function/control key mappings. If the personnel.hlp file does not exist, the program displays a message stating that the help file for personnel form cannot be opened.

```
## activate menuitem "help"
## {
##   help_frs (subject = "personnel form",
##             file = "personnel.hlp")
## }
```

Initialize

Initializes form fields.

Examples—initialize statement:

Example 1:

This example loads data into a form at the start of its display block:

```
## initialize (name = vname, sal = vsal)
```

Example 2:

This example loads data into a form and displays a message:

```
## initialize (ename = "sally", sal = 30000.00)
## {
##   message "you can begin editing data."
##   sleep 2
## }
```

Example 3:

This example illustrates the use of a null indicator variables with a nullable field:

```
## initialize (spouse = spouse_var:null_indicator)
```

Inittable

Associates a table field with a data set.

Examples—inittable statement:

Example 1:

The following example initializes a table field in read mode, with no hidden columns:

```
## inittable empform employee read
```

Example 2:

The following example initializes a table field in the default mode and creates two hidden columns.

```
## inittable empform employee
## (sal = f4, eno = i2)
```

Example 3:

The following example sets display mode depending on program mode:

```
     if (supervisor_mode) then
          mode = "update"
     else
          mode = "read"
     end if
## inittable empform employee mode (eno = i2)
```

Example 4:

The following example initializes a table field and creates a nullable hidden column:

```
## inittable empform employee
##   (spouse = varchar(40) with null)
```

Example 5:

The following example initializes a table field and creates a non-nullable hidden column:

```
## inittable empform employee
##   (salary = money not null)
```

Inquire_frs

Provides run-time information.

Examples—inquire_frs statement:

Example 1:

This example calls a clean-up routine if an error occurred:

```
## inquire_frs frs (err = errorno)
    if (err > 0) then
     call clean_up(err)
    end if
```

Example 2:

This example confirms that user changed some data on the currently displayed form before updating the database:

```
## activate menuitem "update" (validate = 1)
## {
##   inquire_frs form (updated = change)
     if (updated = 1) then
##        getform (newvalue = value)
##        append to newtable (value = newvalue)
     end if
## end
```

Example 3:

This example determines the mode and current field of the form specified by *formname*:

```
## inquire_frs form
##   (modevar = mode(formname), fldname = field(formname))
This example provide a generalized help facility based on the current field name.
## activate menuitem "HelponField"
## {
##   inquire_frs form (fldname = field)
     place appropriate file for fldname into filebuf
##   helpfile "field help" filebuf
## }
```

Example 4:

This example makes sure that the current field in form empform is a table field before deleting the current row:

```
## activate menuitem "Deleterow"
## {
##   inquire_frs field empform (fldname = name,
##                               istable = table)
     if (istable = 0) then
##   message "you must be in the table field to delete ##
            the current row."
##   sleep 2
    else
##      deleterow empform fldname
##  end if
## }
```

Example 5:

This example enables the run-time user to change the row following the current row in a table field, verifies that the current field is a table field and that its next row is visible:

```
## activate menuitem "ChangeNextrow"
## {
##   inquire_frs field "" (istable = table)
     if (istable = 0) then
##        message "you must move to table field"
##        sleep 2
     else
##        inquire_frs table ""
##             (fldname = name, currow = rowno,
##              vlastrow = lastrow)
          if (currow = lastrow) then
##              message "you must scroll in a new row"
##              sleep 2
          else
                currow = currow + 1
           /* update data in row specified by "currow."*/
          end if
     end if
## }
```

Example 6:

The following example inquires whether a field was changed by the run-time user:

```
## activate menuitem "Salary"
## {
##   inquire_frs field (changed = change(salary))
     if (changed = 1) then
          log salary change for employee
     end if
## }
```

Example 7:

The following example updates the database only if the user made changes:

```
## activate menuitem "Update"
## {
    /* check if a change was made to column "rank" in */
    /* the current row. */
    inquire_frs row (changed = change(rank))
    /* only need to update database
    ** if a change was made.
    */
    if (changed = 1) then
         get information and update database
    end if
## }
```

Insertrow

Inserts a new row into a table field.

Examples—insertrow statement:

Example 1:

The following example inserts a new row at the top of the table field display:

```
## activate menuitem "toprow"
## {
##   insertrow empform employee 0 (ename = vname,
##                                  sal = vsal)
## }
```

Example 2:

The following example uses the insertrow statement to enable the user to insert a blank row into the table field before or after the current row:

```
## activate menuitem "InsertBefore"
## {
##   inquire_frs table empform (row = rowno(employee))
     row = row - 1
##   insertrow empform employee row
## }
## activate menuitem "InsertAfter"
## {
##   insertrow empform employee
## }
```

Example 3:

The following example provides a cut and paste facility, using the deleterow and insertrow statements:

```
## activate menuitem "Cut"
## {
##  getrow (vname = ename, vage = age)
##    deleterow empform employee
    cut = true
## }
## activate menuitem "Paste"
## {
     if (cut = false) then
##        message "you must select a row first"
##        sleep 2
     else
##        insertrow empform employee (ename = vname,
##                                     age = vage)
          cut = false
     end if
## }
```

Example 4:

The following example illustrates use of a null indicator variable when assigning a value to a nullable column:

```
## activate menuitem "NewEmployee"
## {
##   insertrow empform employee
##        (spouse = null, title = vtitle:null_indicator)
## }
```

Loadtable

Appends rows of data to a table field's data set.

Examples—loadtable statement:

Example 1:

This example loads data from variables into the ename and eno columns of the employee table field:

```
## loadtable empform employee (ename = vname,
##                              eno = vnum)
```

Example 2:

This example loads all values from the employee database table into the employee table field:

```
##  range of e is employee
##  retrieve (vnum = e.eno, vname = e.ename,
##           vage = e.age, vsal = e.sal,
##           vdept = e.dept)
## {
##   loadtable empform employee
##        (eno = vnum, ename = vname
##            age = vage, sal = vsal,
##            dept = vdept)
## }
```

Example 3:

This example loads information about an employee into a table field

```
## loadtable empform employee
##   (ename = name, spouse = vspouse:null_indicator,
##   manager = null)
```

Example 4:

This example loads the employee table with one row using a loadtable target list:

```
addresses(1) = address_of(vnum)
addresses(2) = address_of(vname)
addresses(3) = address_of(vage)
addresses(4) = address_of(vsal)
load_target_string = "eno = %i4, ename = %c, age = %I4, sal = %f4"
## loadtable empform employee (param(load_target_string,
  addresses))
```

Message

Displays a message on the screen.

Examples—message statement:

Example 1:

The following example displays a message for three seconds:

```
## message "please enter an employee number"
## sleep 3
```

Example 2:

The following example displays a message before initializing two forms. Does not sleep, because the forminit can take a few seconds to complete.

```
## message "initializing forms ..."
## forminit empform, deptform
```

Example 3:

The following example displays a pop-up error message using a buffer:

```
## message errorbuf with style=popup
```

Printscreen

Prints or stores (in a file) a copy of the current form and its data.

Examples—printscreen statement:

Example 1:

This example sends a copy of the form to the printer:

```
## printscreen (file = "printer")
```

Example 2:

This example stores a copy of the current form in the file designated by the filevar program variable, which is prompted for:

```
## prompt ("specify default file for screens: ",
##          filevar)
   if (filevar = "") then
     filevar = "printer"
   end if
## printscreen (file = filevar)
```

Prompt

Prompts the user for input.

Examples—prompt statement:

Example 1:

This example prompts the user for an employee's department:

```
## prompt ("enter the department: ", vdept)
```

Example 2:

This example prompts for a password before opening a database cursor to view and update employee information:

```
## prompt noecho ("enter password: ", passwd)
   if (passwd = "rosebud") then
##   open cursor viewemp
   else
##   message "no permission for task"
##   sleep 2
   end if
```

Example 3:

This example uses a prompt as an interactive message displayed at the top of the screen:

```
## prompt noecho("an error has occurred [press return]",
##               var)
##   with style=popup
##   (startcolumn=1, startrow=1)
```

Purgetable

Clears the list of deleted rows from a table field's data set.

Examples—purgetable statement:

Example 1:

Clear the rows marked for deletion in the data set of the table field employee on the form empform:

```
## purgetable empform employee
```

Example 2:

Clear the rows marked for deletion in the data set of the table field departments on the current form:

```
## purgetable '' departments
```

Example 3:

Clear the rows marked for deletion in the data set of the table field contained in the program variable whichtable, on the current form:

```
## purgetable '' :whichtable
```

Putform

Transfers data into the form.

Examples—putform statement:

Example 1:

This example places data from a constant and a program variable into the form empform:

```
## putform empform (ename = "bill", sal = vsal)
```

Example 2:

This example places data from the database into the current form:

```
## activate menuitem "getnext"
## {
##   retrieve cursor cursor1 (vname, vsal)
##   putform (ename = vname, sal = vsal)
## }
```

Example 3:

This example assigns a null to the description column using the null constant:

```
## activate menuitem "neworder"
## {
##   putform (description = null)
## }
```

Example 4:

This example illustrates the use of null indicator variables when assigning values to nullable columns:

```
## activate menuitem "invoice"
## {
##   /* display salesperson name for invoice, if any */
##   putform (salesperson = name:null_indicator)
## }
```

Example 5:

This example retrieves all the rows from the employee table and displays them one at a time on the form using retrieve and putform param target lists:

```
## retrieve (param(ret_target_string, addresses))
## {
##  putform empform (param(put_target_string, addresses))
##  redisplay
##  prompt noecho ("press return for next row", retvar)
## }
```

Putrow

Updates values in a table field row.

Examples—putrow statement:

Example 1:

In this example, the PutCurrent operation places data in the table field row on which the cursor currently sits:

```
## activate menuitem "putcurrent"
## {
     /* put new information into the current row. */
##   putrow empform employee (age = 52, sal = vsal)
## }
```

Example 2:

In this example, the PutFirst operation puts data in the first displayed row of the table field:

```
## activate menuitem "putfirst"
## {
##   putrow empform employee 1 (sal = vsal)
## }
```

Example 3:

This example illustrates the use of the putrow statement within an unloadtable loop. As the employee table field is unloaded, the putrow statement marks the processed rows to prevent them from being reprocessed. The hidden column marked, specified as i1, is used for this purpose.

```
## activate menuitem "processrecords"
## {
##   unloadtable empform employee
##         (vname = ename, vage = age, vmarked = marked,
##         state = _state)
##  {
    /* process if new, unchanged or changed */
    if ((state = 1 or state = 2 or state = 3) and
       (vmarked = 0))
         then
              process the data
##            putrow empform employee (marked = 1)
         end if
##  }
## }
```

Example 4:

In this example, menu item UpdateOrder takes information about an order from simple fields and updates a table field row with the changed data:

```
## activate menuitem "updateorder"
## {
     /* get data from other fields */
##   getform sales (name:null_indicator = salesperson)
     /* update table field row */
##   putrow sales saleslog
##        (ordernumber = order,
##        salesperson = name:null_indicator)
## }
```

Example 5:

This example illustrates use of the null constant:

```
## activate menuitem "neworder"
## {
##   putrow sales saleslist (salesperson = null,
##                           ordernum = order)
## }
```

Example 6:

Process the records in the data set within a database multi-statement transaction. Reset the _state of any CHANGED or NEW rows so that a user can make further changes to the data set and see those changes correctly processed when this menu item is chosen again. Error handling is ignored.

```
## activate menuitem 'Save'
## {
  ## savepoint startupdate
  ## unloadtable empform employee
     (:ename = ename, :age = age, :eno = eno,
               :state = _state)
  ## {
     if (state = 0) then /* undefined is left alone */
          null
     else if (state = 1) then /* new is appended */
          ## insert into employee (eno, ename, age)
               values (:eno, :ename, :age)

          /* reset _state to UNCHANGED */
          ## putrow empform employee (_state = 2);
     else if (state = 2) then /* unchanged is left alone*/
          null
     else if (state = 3) then /* Reflect changed data */
          ## update employee
               set ename = :ename, age = :age
               where eno = :eno
           /* reset _state to UNCHANGED */
          ## putrow empform employee (_state = 2)
     else if (state = 4) then /* deleted row */
          ## delete from employee
               where eno = :eno
     end if
##   }
##   commit
##   clear field employee
## }
```

Redisplay

Refreshes the screen and displays all visible forms.

Example—redisplay statement:

Example 1:

This example uses the redisplay statement to immediately display the results of the putform statement:

```
## activate menuitem "key"
## {
##   clear field all
##   putform (key = keyvar)
##   redisplay
## }
```

Example 2:

This example retrieves all the rows from the employee table and displays them one at a time on the form using retrieve and putform. After each putform statement, the redisplay statement is issued to display the resulting data, and the user is prompted for the next row.

```
## range of e is employee
## retrieve (vnum = e.empnum, vname = e.empname,
##   vage = e.age, vsal = e.salary)
## sort by #vnum, #vname
## {
##  putform empform (eno = vnum, ename = vname,
##             age = vage, sal = vsal)
##  redisplay
##  prompt ("press q to quit or return continue", resp)
    if (resp = "q") then
##       endretrieve
    end if
## }
```

Resume

Resumes the display loop.

Examples—resume statement:

Example 1:

This example terminates the operation and positions the cursor to the empname field:

```
## resume empname
```

Example 2:

This example terminates the operation and positions the cursor to the menu line:

```
## resume menu
```

Example 3:

This example terminates a field or column activation operation, and completes the cursor movement, FRS key or menu key that initiated the operation:

```
## resume next
```

Scroll

Performs a table field scroll.

Examples—scroll statement:

Example 1:

These operations perform target scrolls to the bottom, top and middle of the data set:

```
## activate menuitem "bottom"
## {
##   scroll empform employee to end
## }
## activate menuitem "top"
## {
##   scroll empform employee to 1
## }
## activate menuitem "middle"
## {
##   inquire_frs table empform (rows = datarows(employee))
     rows = rows/2
     if (rows > 0) then
##         scroll empform employee to rows
     end if
## }
```

Example 2

This example enables the user to find a particular record in the data set:

```
## activate menuitem "find"
## {
    /* prompt for name to search for */
##  prompt ("lastname to search for: ", searchname)
    /* loop through data set and stop when name is found.*/
##  unloadtable empform employee (vname = ename,
##                               record = _record)
##  {
         if (vname = searchname) then
         /* scroll to record with specified name. */
##            scroll empform employee to record
##            resume field employee
         end if
##  }
    /* all rows in data set searched and name not found. */
##  message "cannot find named employee."
##  sleep 2
## }
```

Set_frs

Sets a variety of FRS runtime options and form attributes.

Examples—set_frs statement:

Example 1:

The following statement causes the current field in the current form to appear in reverse video and blink:

```
## set_frs field "" (reverse = 1, blink = 1)
```

Example 2:

The following statement turns off blinking in the partname column of the table field partstbl in the parts form:

```
## set_frs column parts partstbl (blink(partname) = 0)
```

Example 3:

The following statement clears the change variable for a field:

```
## set_frs field empform (change(empname) = 0)
```

Example 4:

The following statement sets the change variable for two columns in a table field:

```
## set_frs row partsform partstable 1
## (change(partnumber) = 1, change(partdescription) = 1)
```

Example 5:

The following example turns on the menu map and sets validation on the menu key:

```
## set_frs frs (menumap=1, validate(menu)=1)
```

Sleep

Suspends execution of the application for a specified number of seconds.

Examples—sleep statement:

Example 1:

The following example displays a message for three seconds:

```
## message "please enter an employee number"
## sleep 3
```

Example 2:

In the following example, if an expert mode flag is set, messages are displayed for a shorter period of time:

```
if (expert_mode) then
     msg_interval = 1
else
     msg_interval = 3
end if
## message "entering information form"
## sleep msg_interval
```

Submenu

Starts a menu that is independent of the current form.

Example—submenu statement:

This example provides a submenu for the Retrieve operation:

```
## activate menuitem "retrieve"
## {
##   submenu
##   activate menuitem "employee"
##   {
     /* select an employee record */
##   }
##   activate menuitem "manager"
##   {
     /* retrieve a manager record */
##   }
## }
```

Tabledata

Steps through the columns in a table field.

Example—tabledata statement:

The following example loops through a form, printing out all field and column names:

```
## formdata formname
## {
##  inquire_frs field "" (fldname = name, istable = table)
    if (istable = 1) then
         print fldname, " is a table field"
##       tabledata
##       {
##            inquire_frs column "" "" (colname = name)
              print colname, " is a column"
##       }
    else
         print fldname, " is a regular field"
    end if
## }
```

Unloadtable

Loops through a table field data set, copying values to host variables and, optionally, executing a block of code once for each row.

Examples—unloadtable statement:

Example 1:

This example processes the records in the data set within a database multi-statement transaction:

```
## begin transaction
## range of e is employee
## unloadtable empform employee
##        (vname = ename, vage = age, vnum = eno,
##         state = _state)
## {
     /* undefined and unchanged are left alone */
     /* new is appended */
     if (state = 0) or
        (state = 2) then
         do nothing
     else if (state = 1) then
##       append employee (empnum = vnum, empname = vname,
##                        age = vage)
     /* reflect changed data */
     else if (state = 3) then
##        replace e (empname = vname, age = vage)
##        where e.empnum = vnum
     /* deleted row */
     else if (state = 4) then
##        delete e where e.empnum = vnum
## }
## end transaction
## clear field employee
```

Example 2:

The following example adds the integer values of number and puts the average in a field called avgvalue:

```
rows = 0
sumvals = 0
## unloadtable statistics accumulator
##  (vname = name, vnumber = number, record = _record)
## {
    if (vname = "") then
##       message "empty numeric identifier"
##       sleep 2
##       scroll statistics accumulator to record
##       resume field accumulator /* break out of loop */
    end if
    rows = rows + 1
    sumvals = sumvals + vnumber
## }
    if (rows > 0) then
         sumvals = sumvals/rows
    end if
## putform statistics (avgvalue = sumval)
```

Example 3:

This example finds the first employee over the age of 60:

```
 found = false
## unloadtable empform employee
##   (vname = ename, vage = age, state = _state,
##    record = _record)
## {
     if (vage > 60) and (state < 4) then /* not deleted */
          found = true
##        endloop
     end if
## }
     if (found) then
          process the specified record
     end if
```

Example 4:

This example illustrates the use of null indicator variables with nullable columns:

```
## activate menuitem "printroster"
## {
##   unloadtable empform employee
##         (vname = ename, vage = age
##         vspouse:indicator_var = spouse)
##   {
         /* do processing to create roster entry. */
##   }
## }
```

Example 5:

This example checks the change variable of rows in a table field data set:

```
## activate menuitem "printorders"
## {
##   unloadtable salesform saleslist
##         (vorder = order, vitem = item)
##   {
            /* check if item ordered has changed. */
##          inquire_frs row salesform saleslist
##               (changed = change(item))
            if (changed = 1) then
                  update the database
            end if
            process order and print out data
##   }
## }
```

Validate

Performs data type checking and field validation checks specified in VIFRED.

Examples—validate statement:

Example 1:

This example validates the data in the field sal:

```
## validate field sal
```

Example 2:

This example validates the data in the field specified by the program variable fieldvar:

```
## validate field fieldvar
```

Example 3:

This example validates the displayed rows of the employee table field before unloading its data set:

```
## validate field employee
## unloadtable empform employee
##   (vname = ename, vage = age, vnum = eno)
## {
##   process the row
## }
```

Validrow

Validates table field rows.

Examples—validrow statement:

Example 1:

This example validates all displayed columns of the current row before issuing a **getrow** statement:

```
## validrow empform employee
## getrow empform employee (vname = ename,
##                           vage = age, vnum = eno)
```

Example 2:

This example validates the sal and age columns of the first displayed row before retrieving the values:

```
## validrow empform employee 1 (age, sal)
## getrow empform employee (vage = age, vsal = sal,
##                           state = _state)
```

Extended Examples

This section contains examples of forms statements in the context of embedded QUEL programs.

Basic Format of a Forms-Based Application

The following example shows the basic format of a typical EQUEL forms-based application program:

```
## agevar  integer
## namevar character_string(20)
## ingres personnel
## forms
## forminit    empform
## display     empform
## initialize  empform (ename = namevar, eage = agevar)
## activate menuitem "Help"
## {
     program code
## }
## activate menuitem "Add"
## {
     program code
## }
## activate menuitem "End"
## {
##   enddisplay
## }
## finalize
## endforms
## exit
```

The display block is the section between the display statement and the finalize statement. The display block contains the statements that display a form and define the operations the user can perform.

The activate menuitem statements create the following menu line:

```
Help Add End
```

Example of Activations

The following example illustrates various activate statements:

```
##  namevar character_string(26)
##  salvar  float
##  ingres "personnel"
##  forms
##  forminit empform
##  display  empform
##  initialize (ename = namevar, sal = salvar)
##  activate field "ename"
## {
     program code
## }
##  activate frskey5
## {
     program code
## }
##  activate menuitem "Help"
## {
     program code
## }
##  activate menuitem "Add"
## {
     program code
## }
##  activate menuitem "End", frskey3
## {
##   enddisplay
## }
##  finalize
##  endforms
##  exit
```

The first activate section is executed when the user moves the cursor out of the ename field. The second section is activated if the user pressed the function, arrow, or control key that is mapped to FRS key 5. The next three activate sections are associated with menu items and cause the following menu line to appear at the bottom of the form:

```
Help Add End
```

Combine any of the conditions in a single activate statement. The user can execute the last activate menuitem section by either selecting the menu item End or pressing the function key mapped to FRS key 3.

Submenus

The following example illustrates the use of a submenu to amplify the Help activate section of the previous example:

```
## activate menuitem "Help"
## {
##   submenu
##   activate menuitem "Application"
##   {
         provide help about current application
##   }
##   activate menuitem "Form"
##   {
         provide help about current form
##   }
##   activate menuitem "MenuOperations"
##   {
         provide help about current menu operations
##   }
## }
```

When the user selects the Help menu operation, this submenu replaces the current menu line:

```
Application Form MenuOperations
```

When the user selects one of the submenu operations, the statements in its activate section are executed. The original menu line then returns.

Nested Menus

Nested menus are invoked using the display submenu statement. The following example shows how the display submenu statement is used to create a nested menu:

```
## activate field "ename"
## {
##   display submenu
##   activate field "empage"
##   {
         do processing for field
##       resume next
##   }
##   activate field "esal"
##   {
         do field processing for field
##       breakdisplay
##   }
##   activate menuitem "end"
##   {
         exit the display submenu
##       enddisplay
##   }
## }
```

Table Fields

The following example illustrates the use of the loadtable statement to load a table field with information retrieved from the employee database:

```
## forms
## forminit empform
## range of d is department
## range of e is employee
   /* get the department name and retrieve information */
## prompt ("enter department name: ", vdept)
## retrieve (vfloor = d.floor) where d.dept = vdept
   /* display the form and initialize fields */
## display empform
## initialize (department = vdept, floor = vfloor)
## {
##  inittable empform employee update (sal = f4, eno = i4)
##  retrieve (vname = e.empname, vage = e.age,
##       vsal = e.salary, vnum = e.empnum)
##       where e.dept = vdept
##  {
##       loadtable empform employee
##            (ename = vname, age = vage,
##             sal = vsal, eno = vnum)
##  }
## }
## activate menuitem "end"
## {
##   breakdisplay
## }
## finalize
## /* update database table – see example below */
## endforms
```

This example prompts the user for a department name, fills the simple fields of the form with information on that department and then loads the table field with rows for the employees in the department. In this example, the loadtable statement is included as part of the initialization of the form's display loop. When the form appears, the table field already contains rows of values. Because the table field is in update mode, users can browse and update the rows in the table field.

Using _record and _state

The following example shows the use of the _record and _state constants in an unloadtable loop:

```
## unloadtable empform employee
##   (vname = ename, vage = age, vsal = sal,
##    vnum = eno, state = _state)
## {
     /* based on the value of "state," the program can
     ** replace, delete or append values in the table
     ** "employee," using "vnum" as the unique identifier
     ** in the search condition. in this example, the
     ** program only performs a replace.
     */
     if (state = 3) then /* state is changed */
##       replace employee
##          (empname = vname, age = vage, salary = vsal)
##           where employee.eno = vnum
     end if
## }
```

Using the Putrow and Getrow Statements

The following example shows activate sections using the putrow and getrow statements:

```
## activate menuitem "putcurrent"
## {
     /* put new information into the current row. */
##   putrow empform employee (age = 52, sal = vsal)
## }
## activate menuitem "getfirst"
## {
     /* get information from the first displayed row. */
##   getrow empform employee 1
          (vage = age, vsal = sal, state = _state)
## }
```

Target Scrolls

The following example contains three activate sections, each of which demonstrates a different use of the target scroll:

```
     /* assume previous declarations, plus ... */
## searchname  character_string(20)
## activate menuitem "bottom"
## {
     /* scroll to end of data set */
##   scroll empform employee to end
## }
## activate menuitem "top"
## {
     /* scroll to first record in data set */
##   scroll empform employee to 1
## }

## activate menuitem "find"
## {
     /* prompt for name to search for */
##   prompt ("name to search for: ", searchname)
     /* loop through data set and stop
     ** when name is found.
     */
##   unloadtable empform employee
##        (vname = ename, record = _record)
##   {
        if (vname = searchname) then
            /* scroll to record with specified name. */
##          scroll empform employee to record
##          resume field employee
        end if
##   }
##   message "cannot find named employee."
##   sleep 2
## }
```

Notice the use of the resume statement in the activate section for menu item find. When the name is found, the activate section is terminated by the resume statement.

Only if all rows are unloaded and the name still has not been found does control pass to the message statement following the end of the unloadtable loop.

Using Resume Statements

In the following example, if the column contains good data, resume next is used to advance the cursor to the next field. If the column does not contain good data, resume returns the cursor to the column:

```
     /* assume previous declarations, plus ... */
## rowcount  integer
## activate column employee ename
## {
##  getrow empform employee (vname = ename)
##  retrieve (rowcount = count (employee.all))
##       where employee.ename = vname
    if (rowcount = 0) then
        /* the name is unique, so continue. */
##      resume next
    else
        /* the name is not unique, so remain on column. */
##      message "employee name must be unique"
##      sleep 2
##      resume
    end if
## }
```

PART 5: 4GL

Chapter 18: Overview of 4GL

This section contains the following topics:

4GL is a *fourth generation language.* With 4GL, you specify what you want done, not how to do it. In contrast to a third generation language such as C, which requires extensive programming, 4GL lets you create forms-based applications with a minimum of programming. Because 4GL applications require fewer lines of code than conventional programs, they are easier to understand and maintain.

How You Can Use 4GL

You can use 4GL with ABF or Vision to customize your applications. For an overview of what you can do, see How the Tools and 4GL Work Together in Overview of Tools and Languages (see page 35).

In general, use 4GL code to implement menu operations, manipulate data in the database, and control the user's movement among the frames and procedures of an application. With 4GL, you can create entry frames and develop customized frames. You can control the sequence of frames in an application as a whole and fine-tune operations such as cursor placement and field validation in a single frame.

You can use 4GL statements to:

- Call other frames, procedures, Ingres user interfaces, or the operating system
- Perform functions such as clearing the screen, displaying error messages, or causing the terminal to beep
- Control what happens when the user chooses a menu operation, presses a specific key, or tries to leave a particular field. You can also specify initialization, time-out, and database event activations. See Frame Activations (see page 810).

4GL provides flexibility in the way you handle data in an application. You can:

- Allow for runtime data input
- Specify operations for querying and updating the database
- Carry out multi-row queries with submenus
- Do conditional processing, such as moving from field to field depending on the application user's data entry
- Perform calculations on items in the frame, whether displayed in the window or not, and pass the results to another frame

Frame Activations

4GL provides the following types of activations you can use to define and control the user's possible courses of action within a frame:

Initialization

An initialization causes a specific operation to occur before the frame is displayed. The syntax for an initialization is:

```
initialize (parameter declarations)
    declare (variable declarations) =
begin
    statements
end
```

Menu Activation

Menus appear on the *menu line*, across the bottom of the frame. They provide options, known as *menu operations*, from which the user can choose. On a user-specified frame, each menu operation invokes processing that you specify with 4GL. Additionally, you can use *submenus* for a variety of purposes.

The syntax for a menu item is:

```
'Menuitem '=
begin
    statements
end
```

Key Activation

A key activation causes a specific action to occur whenever you press a particular key. The syntax for a key activation is:

```
keyname =
begin
    statements
end
```

Field Activation

A field activation causes a specific action to occur whenever the user moves the cursor into a new field or out of a field after changing the value in that field. The syntax for a field activation is:

```
field fieldname =
begin
    statements
end
```

Time-out Activation

A time-out activation causes a specific action to occur whenever the user allows a set period of time to pass without pressing any keys. The syntax for a time-out activation is:

```
on timeout =
begin
    statements
end
```

Database Event Activation

A database event activation occurs when one of the other activations causes the presence of a database event to be detected. The syntax for a database event activation is:

```
on dbevent=
begin
    statements
end
```

Procedures

4GL is generally non-procedural, allowing the application developer and the end user flexibility in choosing which operations take place in a session. However, certain statements can specify procedural operations.

4GL statements can activate the following types of procedures:

- 4GL procedures
- 3GL or host language procedures, written in a standard programming language such as C. The languages supported vary from system to system. For additional flexibility, these procedures can include embedded SQL statements and embedded forms statements.
- SQL or database procedures, written in SQL and encoded in your database.

Chapter 19: Using 4GL

This section contains the following topics:

This chapter introduces the basic syntax you used to create application components, declare their data types, and reference them once they are created. These include:

- Simple application components, such as simple fields and simple local and global variables.
- Complex components, such as table fields, record types, complex local and global variables, arrays, and constants.

For information about specific statement types—including database access and forms-control statements—see Writing 4GL Statements (see page 849). For the full syntax of each 4GL statement, see 4GL Statement Glossary (see page 935).

Form Fields

Form fields (sometimes called form objects) are elements in forms that contain values. Fields are defined for a form using the ABF FormEdit operation. 4GL lets you load, examine and change the values in simple and table fields.

Simple fields, table fields, and derived fields are defined at the time you create the form using the ABF FormEdit operation. For information and directions for creating fields for use in your applications, see the *Character-based Querying and Reporting Tools User Guide.*

Simple Fields

Simple fields hold only a single value at a time and can be visible on the form. An example of a simple field is the Lastname field on a personnel form, which contains only the last name of the employee.

Simple fields can be made invisible to the user, either using the ABF FormEdit operation or dynamically at run time.

Table Fields

Table fields can hold more than one value, allowing you to treat a collection of values as if they were a single component.

Table fields contain rows and columns. Each row of a table field is identical to the others, containing columns with identical configurations of data types. For example, if a row has values of data types varchar(4), float4, and varchar(20), then all rows must have values of the same types, in the same sequence.

A table field, or one or more of its columns, can be made invisible to the user, either using the ABF FormEdit operation or dynamically at run time.

Derived Fields and Columns

A *derived field* or *column* derives its value from a formula. The formula uses the value from another field called the *source field*. When the value in the source field changes, due to user data entry or application action, the value in the derived field or column is recalculated. Neither the user nor the application can directly place values into a derived field or column.

For a value to appear in a derived field, all of its source fields must be valid. Derived fields do not have validations. You can arrange validations for the source fields to indirectly validate the derived field. Values in derived fields are valid only when their source fields are valid, except for aggregates of table-field columns. 4GL always calculates the value in a derived field, even if the result is based only on the default values in the source fields; for example, when you first display a frame.

You can pass values from a derived field or column as from any field or column. However, because a derived field or column can obtain its value only from an expression, you cannot assign values to it. For example, you can pass the value in a derived field in frame A to frame B in a callframe statement, but you cannot use callframe or any other statement to pass the value back to the derived field in frame A.

The way derived fields operate depends on the mode in which the form is displayed. Derived fields are active when a form is displayed in Fill, Update, and Read modes. If the form is displayed in or changed to Query mode, the value of any derived fields is considered unknown. A table field in Query mode is considered to have an empty data set with respect to aggregate calculations.

You cannot directly clear a derived field with the clear statement. However, clearing one of its source fields has the effect of clearing the derived field.

You cannot use the NextField operation (usually mapped to the Tab key) or resume to place the cursor in a derived field.

There are several basic rules to remember when referencing derived fields and columns:

- A derived field or column cannot reference itself (either directly or indirectly).
- A simple derived field cannot reference a table field or column directly.
- A derived column cannot reference a simple field or aggregate value directly.
- A derived field or column cannot reference a local or global variable or a global application constant.
- A simple derived field depends on the aggregate of table-field columns, but not on a particular column in a table field.

For more information on using derived fields, see your query language reference guides. See the *Character-based Querying and Reporting Tools User Guide* for directions for creating derived fields.

Datasets and Form Fields

You create forms and their fields with the ABF FormEdit operation. When 4GL retrieves data from a table in the database into the simple fields or the table field on a form, it stores the rows of retrieved data in a temporary structure known as the *dataset* for the form. Because only a limited number of retrieved records can be displayed in the window at a time, 4GL holds any remaining records in a buffer.

- In the simple fields of a form, one record in the data set is displayed at a time. Succeeding records can be viewed using the next statement with an attached menu.
- In the table field of a form, all records are loaded into the table field. Several records are visible at once in the rows of the table field. You can view any succeeding records of the dataset by scrolling through the table field with the cursor control keys.

Local Variables and Hidden Columns

Local variables and hidden table-field columns contain data that the application manipulates, but are not displayed on the form. These components are useful for holding information that the user does not need. While they are invisible and inaccessible to you, they are accessible to the application in the same way as displayed fields and columns.

Local variables and *hidden columns* are limited in scope to a single frame or procedure, while *global variables* are global in scope for the application. You declare data types for variables as follows:

- Declare the data types of global variables through the frames associated with the ABF Globals menu item.
- Declare the data types of local variables and hidden columns in 4GL statements. This section describes local variable declaration in 4GL specifications.

Declaring Local Variables and Hidden Columns

Local variables and hidden table-field columns can be declared in 4GL in two types of declaration lists associated with the initialize statement. The syntax below shows these statements. *Localcomponent* refers to the local variable or hidden column.

```
initialize [( localcomponent = typedeclaration
        {, localcomponent = typedeclaration } )] =
[ declare localcomponent = typedeclaration
        {, localcomponent = typedeclaration } ]
    begin
        /* statements */
    end
```

Local variables and hidden columns declared within parentheses following the initialize keyword can be accessed by calling frames, which can pass values into them. Local variables and hidden columns declared in the declare section cannot be accessed by other frames.

A 4GL local variable can be of a simple Ingres data type, or a complex user-defined data type (record or array). A hidden column must be of a simple Ingres data type. Local variables of simple and complex types are discussed in the sections that follow.

Data Types for Simple Local Variables and Hidden Columns

A data type specifies the way data is stored in a database table. Examples of data types are integer, char, date, and money. In a declaration statement, you declare a variable or hidden column you are planning to use in your 4GL specification by stating its name and data type. The 4GL syntax for a simple local variable or hidden column declaration is as follows:

```
localcomponent = typedeclaration [ ( length | p,s ) ]
        [ with null | not null ]
```

In the syntax above, *localcomponent* is the simple local variable or hidden column, *typedeclaration* is an Ingres data type name, *length* is the length of a character data type, and *p,s* is the precision and scale for a decimal data type. The length, precision, and scale are all integers. The format for declaring data types in 4GL is the same as that used by the Ingres DBMS in create statements.

4GL does not support the Ingres data types long varchar, long bit, byte, byte varying, and long byte. You cannot define a local variable or hidden column as one of these types. However, if you select a column from the database into an appropriate data type, 4GL coerces and truncates the data. For example, you can select a column of type long varchar from the database into a char or varchar variable.

You can also manipulate long varchars by storing each segment of the long varchar into a varchar element of a 4GL array variable.

4GL recognizes a number of data type names and synonyms. The recommended type names (ANSI standard for character and numeric types) are shown in the following table:

Type	**Length (bytes)**	**Description**
C*n*	1 <= *n* <= *x*	Fixed-length character string, printable characters only. Synonym: c(*n*). *'x'* represents the lesser of the maximum configured row size and 32,000.
char(*n*)	1 <= *n* <= *x*	Fixed-length character string, all characters. Synonym: character(*n*). *'x'* represents the lesser of the maximum configured row size and 32,000.
text(*n*)	1 <= *n* <= *x*	Variable-length character string, all characters except NULL. Synonym: vchar(*n*). *'x'* represents the lesser of

Type	**Length (bytes)**	**Description**
		the maximum configured row size and 32,000.
varchar(*n)*	1 <= *n* <= *x*	Variable-length character string, all characters; *'x'* represents the lesser of the maximum configured row size and 32,000.
float(*n)*	4 if 0 <= *n* <= *7*	Floating-point number. Synonym: f4
	8 if 8 <= *n* <= *53*	Floating-point number. Synonym: f8
decimal(*p, s)*	Depends on precision and scale If precision and scale are not specified, the defaults are (5, 0)	Fixed-point number. Synonyms: dec(*p,s*), *numeric(p,s)*
integer1	1	Integer number. Synonyms: i1
Smallint	2	Integer number. Synonyms: i2, integer2.
Integer	4	Integer number. Synonyms: i4, integer4, int.
Date	12	Abstract.
Money	8	Abstract.

These general definitions apply to the data types:

- An *integer* is a whole number.
- A *floating-point* value consists of an integer, a decimal point, a fraction, and optionally an exponent.
- A *fixed-point decimal* value consists of an integer, a decimal point, and a fraction. Decimal numbers that contain an exponent or that are longer than 31 digits must be treated as floating-point values.
- A *character* value is a sequence of characters.
- The abstract data types *date* and *money* allow you to handle time and monetary units with the facility of the simpler data types.

In SQL, *nullable* versions of each simple data type are allowed. By default, a type in 4GL is nullable unless the not null clause is used. Even so, it is advisable to use the with null clause to clearly document this. For example, to declare the type money nullable, use the format money with null.

For a complete description of each data type, see your query language reference guides. The *Using Character-based Querying and Reporting Tools User Guide* provides more information on declaring data types for variables.

Note for Enterprise Access Products Users

4GL does not support non-OpenSQL data types in database statements in 4GL (select, delete, insert, update). For Enterprise Access products (formerly Gateways), the limits for character data types are as follows:

- char: <= 254;
- varchar: <= 4096

Complex Local Variable Definitions

You can define complex local variables (records or arrays) that are either:

- Of a global record type that you create in ABF or Vision, or
- Based on an existing form, table field, or table.

To define these complex local variables, use one of the following formats:

- *variable* = *record_type_name*
- *variable* = **type of form** *form_name*
- *variable* = **type of table field** *form_name.table_field_name*
- *variable* = **type of table** *table_name*
- *variable* = **array of** *record_type_name*
- *variable* = **array of type of form** *form_name*
- *variable* = **array of type of table field**
  form_name.table_field_name
- *variable* = **array of type of table** *table_name*

In form and table field record types, only visible columns or fields (defined through the ABF FormEdit operation for the form or table field) are used in the record type definition. Local variables or hidden columns (defined in 4GL) are not included. The form or table field on which a record type or array is based need not be used in the current frame, or even in the current application.

Form, table field, and table-based record and array definitions reflect the object at the time the frame or procedure that uses them is compiled. The record or array definition is thus dependent on the form or table. ABF marks the 4GL code for recompilation whenever the form or table changes.

Each of these rules is discussed in the following sections.

Complex Local Variables Based on Record Types

Declare a variable of a complex type as you do a simple local variable, by stating its name and data type. You can base local variables on record types. A local variable defined as a record type or an array references the data in the record or records. For a variable of a record type, use this syntax:

```
variable = record_type_name
```

Record_type_name is a record type defined for the application through ABF. A record type defined for an application overrides any Ingres data type whenever the two conflict.

In the following example, *new_emp* is declared as a variable of the record type *emp_record*:

```
new_emp = emp_record
```

Form-based Record Types

4GL recognizes *form record types*. This record type corresponds to a form you specify; the attributes of the form record type have the same names and data types as the fields of the form. Table fields on the form are included as array attributes of the form record type. Use the following syntax to declare a variable based on a form:

```
variable = type of form form_name
```

Form_name is the name of an existing form. You can base form record types on any form available to the application owner in the application database. They are not restricted to forms in the current application.

In the following example, *empnote_record* is a record based on the form Emp:

```
empnote_record = type of form emp
```

Table-field Based Record Types

In the *table field record type*, the attributes of a record type are given the same names and data types as the columns of the table field you specify:

```
variable = type of table field
  [form_name.]table_field_name
```

Table_field_name is the name of an existing table field on the *form_name* form. Table field-based record types are based on any table field on a form. They are not restricted to table fields in the current form. The table field must be available to the application owner and be in the application database.

The following example creates a variable, *new_emp*, with a record structure the same as the table field *emptable*.

```
new_emp = type of table field empform.emptable
```

In form and table field record types, only columns or fields defined in VIFRED for the form or table field are used in the record type definition. Local variables or hidden columns (defined in 4GL) are not included.

Table-based Record Types

In a *table record type*, you declare a record type that corresponds to a database table. Each attribute of the record type corresponds to a column (attribute) of the database table and has the same name and data type. Use the following declaration syntax:

```
variable = type of table table_name
```

Table_name is the name of an existing database table.

The following example creates a variable, *emp_record*, with a record structure the same as the Employee table.

```
emp_record = type of table employee
```

Arrays Based on Record Types

You can base array declarations on other complex data types. Use this syntax to base an array variable on a record type:

```
variable = array of record_type
```

Record_type is the name of a record type. This syntax cannot be used for hidden column definitions.

The following example creates an array variable, *emp_retire*, with a structure the same as the record type *emp_record*.

```
emp_retire = array of emp_record
```

You cannot define explicit multidimensional arrays in 4GL. To nest arrays, define an array of a record type that includes an array of another record type.

Arrays Based on Forms, Table Fields, and Tables

The syntax for creating arrays based on forms, table fields, and tables is similar to the syntax for creating record types based on these elements. The following examples show the syntax for basing an array on each of these types.

Use this syntax to base an array on a form:

```
variable = array of type of form form_name
```

The following example creates an array variable, *emp_info,* based on the form Employee.

```
emp_info = array of type of form employee
```

Use this syntax to base an array on a table field:

```
variable = array of type of table field
  [form_name.]table_field
```

The following example creates an array variable, *emp_name,* based on the table field Empname in the Employee form.

```
emp_name = array of type of table field employee.empname
```

Use this syntax to base an array on a table:

```
variable = array of type of table  table_name
```

The following example creates an array variable, *emp_arr,* based on the Employee table.

```
emp_arr = array of type of table employee
```

Global Variables and Constants

You can use ABF or Vision to declare global components—variables and constants—to use in an application. You then can use these components in your 4GL code for any frame or procedure.

You can use a global variable or constant in the same manner as any other variable in an application. The only exceptions are that you cannot use global constants:

- On the left side of an assignment
- As the target of a select statement

Always precede a global variable or constant with a colon(:).

When naming global constants and variables:

- Do not use the same name for a global variable and a global constant in the same application.
- Local variable definitions override constant and global variable definitions with the same names.

You can use global constants wherever strings or identifiers are accepted in 4GL, including both the menu item and the explanation in menu item activations. In the following example, the constant *edit_expl* is given a value before the program runs. This string contains an explanation.

```
:edit ( explanation = :edit_expl ) =
    begin
        ...
    end
```

You set values for global variables and constants as follows:

- All global variables are created with an initial value of 0 or the empty string. You must write 4GL code to set a different initial value. You can change the value of a global variable at any point in an application.

  You generally set the initial value of a global variable in the starting frame or procedure of an application, so that it is available to any other frame or procedure that uses the variable. For information on how to verify that you have initialized a global variable before you use it, see the section Using a Procedure to Test Global Variables in Using the Callproc Statement (see page 894).

- You set the value of a global constant when you create the constant in ABF or Vision. You cannot use 4GL to assign a value to a constant or pass it with the byref keyword.

  To change the value of a global constant, you must edit the constant's definition in ABF or Vision. You can create files to store alternate sets of values for constants. For more information on using constants, see the ABF or Vision part of this guide.

Assigning Values to Fields and Variables

Assignment is the process of placing values produced by expressions into fields or variables. You can assign values to fields or simple variables one at a time in direct assignments, or you can assign values to an entire form, table field, record, or array at once in query assignments. This is the basic syntax for assignment:

```
name := | = expression ;
```

- *Name* is the name of the simple field, table field, entire form, or global or local record type, array, or variable to which you are assigning values. *Name* cannot be the name of a derived field.
- "**:=**" or "**=**" are the two forms of the assignment operator. You can use either one. This operator assigns the value of the expression on the right of the statement to the object named on the left.
- A semicolon (;) statement separator is necessary to delimit a statement.
- *Expression* is the value the statement is assigning to *name*. This can be any legal 4GL expression of the same data type as that of *name* or of a compatible data type.

Direct Assignments

A direct assignment statement assigns a value into a simple field in a form, a simple local or global variable, a column in a table field, or the attribute of a record or an array record attribute. The value assigned must have an Ingres data type. For example, you can place a literal value such as "Jones" in the character field Lastname with either of these assignment statements:

```
lastname := 'Jones';
lastname = 'Jones';
```

In the following example, the statement is illegal because the left- and right-hand sides of the assignment have incompatible types. Lastname is a character field while 35 is an integer.

```
lastname := 35; /* error! */
```

You can make simple assignments into a table field cell. The following statements place the specified values in the current row of the table field:

```
child.name := 'Steven';
child.age := 11;
```

Except for unloadtable loops, the current row is the row on which the cursor is positioned. A runtime error occurs if the cursor is not positioned with the table field.

The following statements assign the specified values to the second row in the *child* table field:

```
child[2].name := 'Sally';
child[2].age := 8;
```

Note that the integer expression "2" refers to a row in the table-field display in the window, not to a record in the underlying data set.

To make a direct assignment of values to record attributes, place the name of the record and attribute (in *record.attribute* format) on the left of the assignment statement. This example assigns the specified value to the *name* and *salary* attributes of the record *employee*:

```
employee.name := 'John' ;
employee.salary := '50000' ;
```

Assigning values to arrays is similar to assigning values to table fields. Place the name of the array, index, and attribute on the left of the assignment statement and the value on the right. Arrays differ from table fields in that, for arrays, the index is always required, except in an unloadtable loop.

This example assigns the specified value to the *name* and *address* attributes in the third record of the array named *parent*:

```
parent[3].name := 'Janet' ;
parent[3].address := 'New York' ;
```

The "3" in brackets is an integer expression that indicates the third record of the array.

Assignments Using a Query

The query assignment statement assigns a series of values from the database to several or all simple fields on a form, to the columns of a table field or table field row, or to the attribute of an array or record. In a query assignment, you use a select statement. You can optionally include a submenu.

Query Assignment to a Form

A query assignment statement to a form uses a query to assign a list of values from the database to one or more simple objects—a simple field on the form, a simple local variable, or a simple component of a complex object. Query targets for query assignment to a form are the same as those for direct assignment. For this type of assignment, specify the *form* name instead of a *field* name on the left of the assignment statement. For example:

```
empform := select projname, hours
    from projects;
```

In this case, *empform* is the form associated with the frame. It contains simple fields corresponding to *projname* and *hours* in the Projects database table. The select statement is the query that causes Ingres to read the data into the fields on the form.

The target simple objects can have names that are different than the database columns, but they must match in data type. If the names are different, you must use both of them in the assignment statement as shown below. The following statement retrieves values from Projname to the Task field, while the values from Hours are retrieved to the Time field.

```
empform := select task = projname,
    time = hours
    from projects;
```

As a simple field assignment, this statement reads one record from the database table.

If you specify both a target simple object and database column, then you can precede the target simple object with a colon (:). If the target simple object is a simple component of a complex object, you must specify the target object by a qualified name and precede it with a colon (:).

Query Assignment to a Table Field or Array

In a query assignment to a table field or array, the query targets are the array's attributes or the table field's columns. If the query is successfully executed, the previous contents of the table field or array are cleared, and the results of the query become the new contents. By default, the table field or array is cleared even if the query does not return any rows. Retain the previous contents if the query does not return any rows. Use **set_4gl (clear_on_no_rows=off)** to turn off the default behavior. See Set_4gl (see page 1132).

In the following example, the select query reads multiple records selected from the database table *parts* into a data set associated with the table field *partstbl*.

```
partstbl := select number, name, price
    from parts;
```

This example might result in the display of the following *number*, *name* and *price* items:

2445	Socket wrench set	$49.99
2446	Socket wrench set	$62.99
2562	Screwdriver	$ 8.29
2563	Screwdriver	$ 5.99
2565	Screwdriver	$ 9.99
2566	Screwdriver	$12.99

4GL responds by creating a data set from the results of the query, and displaying as many rows as the table field can accommodate, as defined through the ABF FormEdit operation. You can scroll through all the records in the data set by using the mapped cursor keys to display subsequent rows or by scrolling programmatically.

Query Assignment to a Record, Table Field Row, or Array Record

In a query assignment to a record, table field row, or array record, the query targets are the attributes of the array or record or the table field's columns. This type of assignment is a singleton query; it reads one row from the database.

If the select statement is executed successfully, all columns or attributes are cleared before the values are returned from the database. Default values (zeroes or blanks) are placed into any column or attribute for which no value is returned.

To select values into specific columns or attributes without clearing others, use the syntax for selecting to simple objects (as shown in the last example in this section), instead of specifying a query assignment to the record, table field row, or array record.

If the select statement does not return any rows, do not clear all the columns or attributes. By default, the columns or attributes are cleared even if the query does not return any rows. However, you can use **set_4gl (clear_on_no_rows=off)** to turn off the default behavior. See Set_4gl (see page 1132).

The examples below show similar query assignments to an array record and to a table field row. Note the differences in syntax.

```
emptbl[]:= select name, job_title,
        ss = ssno
    from employees
    where ssno = '555-55-5555';
```

The columns *name*, *job_title*, and *ssno* are read from the Employees table for the record whose value for *ssno* is "555-55-5555." They are read into the table field columns *name*, *job_title*, and *ss* for the current row of the table field. The cursor must be in the table field for this query to succeed. This row of *emptbl* has another attribute, *salary*, which is cleared.

```
emparr[1] := select name, job_title,
        ss = ssno
    from employees
    where ssno = '555-55-5555';
```

In this example, the columns *name*, *job_title*, and *ssno* are read from the Employees table for the record whose value for *ssno* is '555-55-5555'. They are read into the attributes *name*, *job_title*, and *ss* for the first record in the array *emparr*. This array has another attribute, *salary*, which is set to 0.

The following example shows assignment from a query to attributes of an array. It reads the columns *name*, *job_title*, and *ssno* without clearing the *salary* column.

```
select :emparr[1].name = name,
         :emparr[1].job_title = job_title,
         :emparr[1].ss = ssno
    from employees
    where ssno = '555-55-5555';
```

The columns *name*, *job_title*, and *ssno* are read from the Employees table for the records whose value for *ssno* is "555-55-5555." They are read into the attributes *name*, *job_title*, and *ss* for the first record in the array *emparr*. The other attributes in *emparr* retain their previous values.

The select statement is described further in Select.

Assigning Nulls to Nullable Variables

4GL provides the following three ways to assign a null value to a nullable variable:

Direct assignment

To assign a null to a simple field or a local or global variable, use the format:

```
field := null;
```

To assign a null to an attribute of a record, use the format:

```
recordname.attributename := null;
```

To assign a null to a table-field item, use the format:

```
tablefieldname[row].columnname := null;
```

From the database

If you select a column from the database that contains a Null value, the receiving 4GL variable contains NULL. The variable must be of a nullable type.

From a system function

If a system function does not return a valid value, the receiving variable is set to NULL. For more information about system functions, see System Functions in Using Procedures as Expressions (see page 836).

Through the FRS

A field is assigned a Null value in the following cases:

- If you tab through a nullable field or nullable table-field column without making an entry.
- If you press Return without entering a value in response to the prompt statement (and the receiving variable is nullable).

Referencing Fields, Variables, and Constants

4GL statements reference fields and variables to:

- Retrieve data from the database into fields and variables
- Insert/update data to the database from fields and variables
- Assign values to/from fields and variables
- Use fields and variables in expressions
- Pass fields and variables as parameters to other frames and procedures

Referencing Simple Fields

Reference simple fields by referring to the field's *internal* name as defined through the ABF FormEdit operation. Fields cannot be referenced using the field's title as displayed on the form.

Use the name of any simple field in expressions or assignment statements, as in this example:

```
total := valuea + valueb;
```

Here *total*, *valuea* and *valueb* are all (internal) field names on a form. If there is any chance of confusing the field name with literal strings, use a colon:

```
total := :valuea + :valueb;
```

Observe the following rules for using a colon for dereferencing a simple field:

- Do not include a colon when assigning a value to a field; that is, the field appears on the left-hand side of the assignment.

  (However, the colon is allowed when a simple field appears on the left-hand side of an assignment within the target list of a select into a form or a select into simple objects.)
- The colon is optional if the field appears elsewhere; for example, on the right-hand side of the assignment.

  (However, the colon is required if you are using the field as a dynamic 4GL name or to specify dynamically a value in a DBMS statement.)

Referencing Local and Global Variables

Reference a local or global variable in 4GL to access a simple field.The result of changing a local or global variable or hidden column is not displayed in the window.

Hidden elements are often used for storing the temporary results of calculations that 4GL performs. Here is an example:

```
total := total + current;
```

In this example *total* is a visible field on a form that is adding up values, but *current* is a local variable that stores a subtotal to be added to the previous total.

While you can declare global variables using ABF frames associated with the Globals menu item, you declare local variables and hidden columns in a 4GL specification. The name of a local variable cannot correspond to the internal name of any field on the current form.

Referencing Table Fields, Columns, and Hidden Columns

The procedure for referencing a table field, table field column, or hidden column in 4GL is similar to the procedure for referencing a form containing simple fields. If you do not name the row number explicitly, 4GL assumes you mean the column in the *current row*—the row on which the cursor is currently located.

If the cursor is not currently on the table field, the current row is undefined, except in an unloadtable loop. Data retrieval at run time is unsuccessful.

The syntax for referencing a table-field column or hidden column in the current row is:

```
tablefieldname.columnname
```

The colon, if used, must precede the full designation of the column, as in the following example:

```
callframe :tablefieldname.name;
```

Follow the rules for simple fields stated earlier in this section to determine when to include the colon with a column name. The only difference is that the colon is required when a table field column appears on the left-hand side of an assignment within the target list of a select into a form or a select into simple objects.

Statements referring to a column always refer to the column's *internal name*, as defined using the ABF FormEdit operation. This name is distinct from the title above the column on the form.

To reference the column *name* in the table field *children* for the current row, use:

```
children.name
```

The following statement assigns the value "Grey" to the *name* column of the current row:

```
children.name := 'Grey';
```

How You Reference an Element of a Table Field

To reference individual elements of a table field, you can specify or index a row by number and a column by name. The syntax for referring to a column is:

```
tablefieldname [integerexp].columnname
```

Integerexpr is an expression that evaluates into an integer, and is not nullable. It is the number of a row the table field displayed in the window, not the record number in the underlying data set.

For example, the following statement assigns the value "Sally" to the *name* column in the third row of the *children* table field:

```
children[3].name := 'Sally';
```

Because the row number is specified, this statement is valid whether or not the cursor is on the table field.

Referencing Records

The syntax for referencing record attributes is similar to the syntax for accessing values in a table field:

```
record_identifier.attribute_identifier
```

If an attribute of a record is also a record, you can qualify it further by using a subattribute name to access the value it contains. The 4GL compilers check the name to make sure that the attribute type contains the named attribute. The following example references the attribute *id* in the record *dba*, which is itself an attribute of the record *session_info*.

```
session_info.dba.id
```

Referencing Arrays

The syntax for referencing the values in arrays of records is similar to that for table field columns. Unlike with table fields, there is never a current record in an array, except in an unloadtable loop (discussed in Unloadtable).

You can reference the values in arrays by placing the index number inside square brackets [ ] . Use this syntax:

```
array_identifier [integer_ref].attribute_identifier
```

When referring to array records by index position, remember that if you insert or delete a record in the middle of an array, the index numbers of the subsequent records are changed.

The following example sets the Name attribute of the second record of the employee array:

```
employees[2].name = 'Jones'
```

In the following example, an array is declared to be of type of form "empform." The form contains the table field "dependents." The example sets the Dependent Name attribute for the third record of the array (which contains the second row from an array based on the table field):

```
a_empform[3].dependents[2].dep_name= 'ralph';
```

Scope of Reference Statements

Most statements that reference fields and table fields are local in scope and apply to the form associated with a particular frame. Two statements used for passing data to other frames and procedures allow you to reference fields in other frames. You can also use these statements to reference local variables declared as parameters in the initialize statements of other frames.

- The callframe statement, used to pass data to another frame, allows you to pass data from fields and local variables to another frame.
- The callproc statement, used to pass data to another procedure, allows you to pass data from fields and local variables to a procedure.

You can reference global variables and constants from any frame or 4GL procedure. 3GL procedures can reference any form after it has been displayed.

Expressions

In 4GL, an *expression* can determine the value of a form component or can be a condition that can be evaluated to "True" or "False." An expression *contains* values, *is* a value, or *produces* values through processing.

An expression consists of a combination of operators, such as logical or arithmetic operators, and operands, which can be of many types. Expressions make up statements, which in turn make up 4GL specifications. All the operators and functions of the DBMS are available for use in 4GL expressions. These expressions include:

- Literals and constants (named literals)
- Simple fields and table-field columns
- Local and global variables
- Records and arrays based on forms, table fields, and tables
- Database access through query statements
- Procedures and values returned by procedures
- Numeric and string expressions
- Logical expressions

You can use expressions to substitute values for processing while your application is running.

Using Literals in Expressions

The following literals are valid expressions:

```
'J. J. Jones'
'Hendersonville'
'17-Aug-1998 10:00'
1209
7.77
```

4GL recognizes two basic types of literals, string and numeric. In addition, it recognizes the special literal NULL.

- *String literals* include character strings, date, and hexadecimal strings.
- *Character literals* are represented by a sequence of characters enclosed in single quotation marks. Within a string literal, you can indicate a literal single quote by typing two single quotes (for example, 'J. J. Jones''s').
- *Application constants* are represented by the constant name preceded by a colon (for example**, :help_opt**).

- *Date literals* must be specified as strings (for example, '11/15/98') but you can manipulate dates in date arithmetic operations.
- *Hexadecimal numbers* are represented by a special form of string literal. This is a string of hexadecimal digits (0 through 9 and A through F) preceded by the letter X and enclosed in single quotes. For example:

  ```
  X'10A665B'
  X'00FF'
  ```
- *Numeric literals* include integers and floating-point numbers. You can specify literals of type money as strings or numbers (for example, '$10.50' or 10.5).

How You Use Hexadecimal for Nonprintable Characters

You can represent printable characters literally as strings. To specify a nonprintable character, use a hexadecimal literal of the form:

X'nn{nn}'

In this form, each character is represented as two hexadecimal digits (*nn*). For example, the following statement inserts the string "XYZcarriage return" into "col1" and a numeric into "col2" of "table1":

```
insert into table1 (col1, col2)
    values (X'58595A0D', 500);
```

The hexadecimal literal is translated to the corresponding character value.

Using Fields and Variables as Expressions

Each field or variable in a form can be used in an expression or as an expression in itself. In this example, the "age" field appears alone on the left side and is an element in the expression "age + 1" on the right of the assignment statement:

```
age := age + 1;
```

The current value is used in the computation of a new value that increments and replaces the current value.

In the same way, you can use a column or hidden column in an expression or as an expression in itself:

```
total := 0;
unloadtable emptable
begin
    total := total + emptable.salary
end
```

In this example, the "salary" column from the "emptable" table field is added to the "total" field as part of an unloadtable loop.

Using Procedures as Expressions

When a procedure returns a value such as a return code, it functions as an expression, as in the example below:

```
returnval := procname();
```

Procname is the name of a procedure that returns a value. If you are not passing any parameters to the procedure, use an empty set of parentheses, as shown above.

A procedure can also be used in an expression. A procedure named mod() is used in the following example:

```
if mod(flag/256, 2) = 1 then
    message 'bit 8 set'
endif
```

System Functions

4GL provides a set of implicit system functions for use in your applications. These include the ifnull function and the following scalar functions (that is, functions that accept single-valued expressions, rather than sets, as their arguments):

Type conversion functions

The following functions set the data type of an expression: c, char, date, decimal, dow, float4, float8, hex, int1, int2, int4, money, numeric, text, varchar, table_key, object_key

Numeric functions

The following functions perform mathematical operations on an expression: abs, atan, cos, exp, log, mod, sin, sqrt

String functions

The following functions operate on character strings: concat, left, length, locate, lowercase, pad, right, shift, size, squeeze, trim, notrim, uppercase, charextract

Date functions

The following functions operate on date values: date_trunc, date_part, date_gmt, interval, date, _time

The following examples demonstrate some of the ways you can use system functions.

- Convert a number in the integer variable "age" to a character string so that the value can be used in a message statement:

```
message 'average age is: ' + char(:age);
```

- Retrieve the current date:

```
curdate := date('now');
```

- Calculate the number of days between a date and today:

```
number_of_days := interval ('days',
start_date - 'today');
```

- To return into a varchar variable the current time on the machine on which an application is running (the time is expressed as the number of seconds since January 1, 1970):

```
var = varchar(int4(interval('sec',
'now'-date('1970.01.01 00:00:00gmt'))));
```

  If the function returns an error, the variable is set to NULL. If you use a function to assign a value to a variable, use error checking to ensure that the function returned a valid value.

See the *SQL Reference Guide* for detailed descriptions of the system functions listed above.

You can use the aggregate functions (that is, functions that operate on a column of values, such as count) only within a database access statement, such as a select. This restriction also applies to the dbmsinfo function that retrieves system information. See Using the Dbmsinfo Function for more information about using the dbmsinfo function.

ABF displays a warning message if you define a procedure with the same name as a system function. If you decide to override the system function name, your procedure overrides the system function in the current application.

This is not the case with query expressions. System functions are the only procedures recognized inside query expressions. This means that a system function overrides your procedure only within a query expression.

How Numeric Expressions Work

The arithmetic operators in 4GL combine numeric expressions into new numeric expressions.

Operator	**Description**
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

In addition to standard numeric usage, you can do date arithmetic. For example, the following statement computes the result value for the date field "start_date":

```
start_date := start_date + '2 days';
```

Precedence of Arithmetic Operators

Arithmetic operators have the following precedence (in descending order):

```
**
or /
+ or -
```

Precedence is handled as follows:

- In the absence of parentheses:
  - Operators of higher precedence are done first.
  - For adjacent operators of equal precedence, operators are done from left to right.
- If there are any operators within parentheses, these are done, beginning with the innermost parentheses, before any operators outside of parentheses.

You can use parentheses to change the default order of operators, or simply to make the order explicit; for example, assume you have the expression:

```
a - b + c / d * e ** f
```

You can use parentheses as follows to emphasize the default order:

```
(a-b)+((c/d)*(e**f))
```

Use parentheses as follows to change the order:

```
(a-(b+(c/d))*e)**f
```

String Expressions

The concatenation operator (+) joins string expressions together into new expressions. For example, the following statement builds a prompt using literals and a variable:

```
answer := prompt
    'Please enter department for ' +
        :name + ': ';
```

Logical Expressions

Logical expressions are expressions yielding the Boolean values True, False, or Unknown (for NULL values). 4GL includes the Boolean comparison operators and the logical operators AND, OR, and NOT.

Boolean Comparison Operators

The Boolean comparison operators join numeric or string expressions into a logical expression. The following table summarizes the Boolean comparison operators:

Operator	Description	Left-hand Operand	Right-hand Operand
=	Equal to	Numeric or string expression	Numeric or string expression
!= <> ^=<	Not equal to	Numeric or string expression	Numeric or string expression
<	Less than	Numeric or	Numeric or string

Operator	Description	Left-hand Operand	Right-hand Operand
		string expression	expression
<=	Less than or equal to	Numeric or string expression	Numeric or string expression
>	Greater than	Numeric or string expression	Numeric or string expression
>=	Greater than or equal to	Numeric or string expression	Numeric or string expression
IS NULL	Value is "null"	Numeric or string expression	None
IS NOT NULL	Value is other than "null"	Numeric or string expression	None
IS INTEGER	Value is an integer	Numeric or string expression	None
IS NOT INTEGER	Value is other than an integer	Numeric or string expression	None
IS DECIMAL	Value is a decimal type	Numeric or string expression	None
IS NOT DECIMAL	Value is other than a decimal type	Numeric or string expression	None
IS FLOAT	Value is a float type	Numeric or string expression	None
IS NOT FLOAT	Value is other than a float type	Numeric or string expression	None
LIKE	Value matches pattern-matching string	String expression	See The LIKE Operator and Pattern Matching
NOT LIKE	Value does not match	String	See The LIKE

Operator	Description	Left-hand Operand	Right-hand Operand
	pattern-matching string	expression	Operator and Pattern Matching

The result of a comparison with a nullable value can be True, False or Unknown. The result is Unknown if one or both operands of the expression are NULL, except for the IS NULL and IS NOT NULL operations.

4GL tests the resulting expressions in if and while statements. 4GL also uses them in the where clauses of query statements.

The condition that a 4GL if or while statement tests must be a Boolean expression. For example, if Status is a character field in the form currently being displayed, then the following condition yields a Boolean value:

```
status = 'n'
```

The **if** statement below leads to a call on NewProject when the Status field has "n" as its value:

```
if status = 'n' then
    callframe NewProject;
endif;
```

In if and while statements, if the result of an expression is *Unknown*, then the flow of control occurs exactly as if the Boolean expression evaluates to False. For examples, see Nulls in Expressions (see page 845).

The LIKE Operator and Pattern Matching

4GL allows you to compare two strings to see whether they resemble each other in specific ways.

To do this, specify a *pattern*—a string of characters with special formatting characters—that shows what the compared string must look like, along with the LIKE pattern-matching operation. If the string you specify is an instance of the pattern string, then the comparison evaluates to True. The syntax for the LIKE operation is as follows:

```
variable [not] like pattern [escape escapechar]
```

Variable is the name of a character string field or variable, *pattern* is a character string literal or character string variable, and *escapechar* is a character string literal or character string variable of length 1. *Pattern* must not be NULL.

Pattern strings can include the following special characters:

- An underscore (_) is a "wild card" that matches any single character. For example, "Xa," "aa" and "/a" are all text strings that match the pattern string "_a."
- A percent sign (%) is a "wild card" that matches any string of characters, regardless of length. For example, the string "Fred%" can retrieve the pattern strings "Fred," "Frederick," and "Fred S. Smith, PhD."
- Square brackets [ ] when preceded by an *escape* character match the corresponding character position with any of the characters within the bracketed string. Examples are given below with a further description of this feature.

In comparisons using the arithmetic operators, pattern-matching characters do not have any effect. For example, in the following statement, the comparison tests the value of *name* to see whether it exactly equals the constant literal "Leslie%." If it does, the object of the conditional is executed.

```
if name = 'Leslie%' then ...
```

To make the pattern-matching characters take on their special meanings, use the LIKE operator as any other comparison operator is used. For example, the following statement tests whether a text value ends in the letter *e*:

```
if name like '%e' then
    message 'Found a name ending in e';
endif;
```

The following statement tests the value of variable *emp_name* against the pattern, to see if it starts with "Leslie", and it makes sure that the last name of the employee is "Smith":

```
if emp_name like 'Leslie%Smith' then
    message 'Found a name matching the pattern'
endif;
```

You can use the escape clause with the LIKE operator in two ways.

- Use the escape clause to "escape" the special interpretation of the pattern matching characters "_" and "%." For example, the following phrase matches a string with any first character and an underline as the second character:

  ```
  name LIKE '_\_' ESCAPE '\'
  ```

  The next phrase matches any string beginning with a period (.) and ending with a percent sign:

  ```
  name LIKE '..%.%' ESCAPE '.'
  ```

- Use the escape clause to give a special interpretation to square brackets [ ] in the pattern string. Normally, square brackets are treated like any other characters. However, when preceded by the escape character, the brackets "escape" their standard interpretation and can define a match-any-of-these-characters string.

For example, suppose you want to match all strings ending with X, Y or Z. Use the following phrase:

```
name LIKE '%\[XYZ\]' ESCAPE '\'
```

In the following example, the pattern matches "ABC," "ACC," and "FCC Fairness Doctrine," and does not match "FDC Yellow #42" or "Access."

```
name LIKE '_\[BC\]C%' ESCAPE '\'
```

The next example matches any string beginning with a left bracket and whose second character is "1," "2," "3" or a right bracket:

```
name LIKE '[\[123]\]' ESCAPE '\'
```

An **escape** character must be followed in the pattern by an underscore, percent, left or right bracket, or another escape character.

AND, OR, and NOT Operators

The logical operators (AND, OR, and NOT) join logical expressions into new logical expressions. The following truth tables show the result of comparisons made with the AND, OR, and NOT operators:

The following table lists the AND operators:

	True	**False**	**Unknown**
True	True	False	Unknown
False	False	False	False
Unknown	Unknown	False	Unknown

The following table lists the OR operators:

	True	**False**	**Unknown**
True	True	True	True
False	True	False	Unknown
Unknown	True	Unknown	Unknown

The following table lists the NOT operators:

True	**False**	**Unknown**
False	True	Unknown

The example below illustrates the use of the IF and AND operators:

```
if empnum > 0 and status != 3 then
    callframe newemployee;
endif;
```

The frame *newemployee* is called only if both conditions are True. The current value of *empnum* must be greater than zero and *status* must have any value other than 3.

This example illustrates the use of the NOT and OR operators:

```
if not (status = 1 or status = 2) then
    callframe newemployee;
endif;
```

The frame *newemployee* is called if neither of the conditions are True. The current value of *status* cannot be 1 or 2.

Precedence of Logical Operators

The logical operators have the following precedence (in descending order):

NOT
AND or OR

Precedence is handled as follows:

- In the absence of parentheses:
  - NOT is done before AND or OR.
  - Adjacent occurrences of AND and OR are done from left to right.
- If there are any operators within parentheses, these are done, beginning with the innermost parentheses, before any operators outside of parentheses.

You can use parentheses to change the default order of evaluation, or simply to make it more explicit; for example, the following expression:

```
x = a OR x = b AND NOT y = c
```

is equivalent to:

```
(x=a OR x=b) AND (NOT y=c)
```

The order of precedence is different for 4GL logical operators than for SQL logical operators. To avoid confusion when using logical operators in 4GL, use parentheses in expressions that use a mixture of the operators.

Nulls in Expressions

Because the NULL value cannot be compared to another value, the only test you can perform is to check whether a value is NULL or not. To test this, use the IS NULL and IS NOT NULL operators in a conditional statement. The syntax is similar to that of any other comparison operator:

```
[expression] is [not] null
```

Here is an example:

```
if salary is null then
    message 'Salary amount is unknown.';
endif;
```

Nulls in expressions follow these general rules:

- If any item in an expression has a NULL value, then the value of the entire expression is NULL.

  In the following example, if the variable *empno* has the value Null, then *msg* has the value Null after the statement is executed:

  ```
  msg := char (empno) +
      ' is not a valid employee number';
  ```

- If any of the variables contain a Null value, then the result of any comparison involving them is Unknown.

  For example:

  ```
  count := null;
      if count + 1 > 0 then
          callframe newproject;
      endif;
  ```

  Because *count* is Null, the result of the comparison that includes *count* is Unknown; hence, the **callframe** statement is not performed.

  This rule holds true for more complicated expressions, as in the following statements:

  ```
  man_days := char(days) + ' days';
  if (start_date + man_days) > 'today' then
      statements
  endif;
  ```

  If *start_date* or *man_days* has a Null value, then the entire Boolean expression evaluates to Unknown. Again the "*statements*" are not performed.

4GL Names

In 4GL, you can specify most of the names and identifiers in statements either statically (when the application is written) or dynamically (at run time). Names that you can specify in both ways are known as *4GL names*. The 4GL name is one of the means 4GL provides for substituting values for processing while your application is running.

4GL names include most elements of 4GL syntax that are not reserved words or expressions. Categories of 4GL names include:

- Names of form objects, such as fields and table field columns, in forms statements (such as clear, scroll, resume, and validate)
- Names of database objects, such as tables and table columns, in most contexts within DBMS statements
- Parameter names
- Miscellaneous words in 4GL statements, such as the *procedurename* in a callproc statement

A 4GL name can be:

Static

- A character string optionally enclosed by quotation marks (the quotation marks are required if the character string is identical to a reserved word)
- An integer.

Dynamic

- Any of the following simple objects (preceded by a colon): a simple field, a simple local variable, a simple global variable, a global constant, or a simple component of a complex object
- A simple variable or constant name, optionally preceded by a colon

For example, in the following example, *framevar* must be a simple field or variable or global constant of string type:

```
callframe :framevar
```

Avoid using a variable of a nullable type as a 4GL name, because a 4GL name cannot contain a null value. If the statement that contains this error executes, the value returned is zero if a number is expected or "$NULL$ERROR" if a character string is expected.

Note: In this guide, 4GL names are underlined in syntax statements.

Examples

The examples below illustrate the use of 4GL names and expressions.

In the following example, "nextframe" is understood as a literal because no colon precedes it and because the callframe parameter is always a 4GL name.

```
callframe nextframe ;
```

You can use a quoted string in a 4GL name; the quotation marks do not affect the interpretation of the statement; the next statement has the same effect as the first statement.

```
callframe 'nextframe' ;
```

The name "frametocall" in the example below is the name of a field on a form; thus, this is a field assignment. Neither side is a 4GL name. The word "nextframe" is an expression (a string literal) and therefore must be quoted. Without quotes, 4GL expects to substitute a value from a field or variable named 'nextframe.'

```
frametocall := 'nextframe';
```

In the following example, "Lastframe" is a field or variable containing a valid frame name. These are field assignments with substitution intended. Either is correct because the right-hand side is always considered an expression.

```
frametocall := lastframe ;
frametocall := :lastframe ;
```

The following example uses a 4GL name to allow substitution of a value from the specified field. The colon is required; otherwise the string is interpreted as a literal, as in the first example above.

```
callframe :frametocall ;
```

In this statement, "frametocall" is a field name. Its value is "nextframe," the name of a frame in the application. Because it is not prefixed with a colon, callframe attempts to call a frame named "frametocall," generating an error.

```
callframe frametocall ;    /* error */
```

Chapter 20: Writing 4GL Statements

This section contains the following topics:

This chapter introduces basic techniques for writing simple 4GL statements for your applications. Sample 4GL statements are presented as an elementary roadmap. The chapter concludes with a complete example of a frame specification.

The techniques and information introduced here build on the discussion of forms, fields, and data types presented in Using 4GL (see page 813).

For additional suggestions about creating applications, see Sample 4GL Application (see page 1187), which describes a sample application in detail. In addition, see the ABF part of this guide.

For the full syntax of each 4GL statement, see 4GL Statement Glossary (see page 935).

Basic Statement Types

For each operation in a 4GL specification, you specify one or more 4GL statements. 4GL includes these basic types of statements:

- Declaration statements
- Forms-control statements
- Database access statements
- Flow-control statements
- Call statements
- File access statements

- Inquiry statements
- Session connection statements

Each operation consists of a specific sequence of these statements between the keywords begin and end or between a pair of braces.

Separate individual 4GL statements with semicolons, as in this syntax:

```
callframe newframe;
```

For parameter lists in callframe and callproc and local data declarations, you can use a semicolon or a comma as a separator. For example, in the where clause of the statement below, you can use either a semicolon or a comma after the first expression.

```
callframe empframe (empframe.emptbl =
        select * from emp
        where mgr := :mgr, dept := :dept);
```

In addition, 4GL accepts an empty declaration so that you can terminate the list of declarations with a comma or semicolon. This following example contains an empty declaration after the comma:

```
initialize ( ) =
        declare
            x = integer not null,
        { }
```

Using Initialize and Declare Statements

Use declaration statements during the initialization operation to declare local variables or hidden columns in a table field on a form. Local variables and hidden columns are useful for computation and holding temporary data that the user does not see or change.

Two types of declaration lists are associated with the initialize statement. The following syntax shows these lists:

```
initialize
     [( localvariable | tablename.hiddencolname =
      typedeclaration {, localvariable |
      tablename.hiddencolname = typedeclaration} ) ] =
[declare localvariable | tablename.hiddencolname =
      typedeclaration {,localvariable |
      tablename.hiddencolname = typedeclaration }]
begin
        /*  statements * /
end
```

- Local variables and hidden columns declared within the parentheses following the initialize keyword can be parameters to the frame. They can receive values from other frames through call parameter lists by keyword.
- Local variables and hidden columns declared in the declare section cannot receive values from other frames through call parameter lists.

A 4GL local variable can be of any data type. This includes simple Ingres data types, records derived from form, table field, or table definitions, record types defined to the application, and arrays of records.

For clarity, place all local variables in the declare section to distinguish them from passed parameters. The following example establishes the variable *i* as an array index that is strictly local to the current frame:

```
initialize () =
declare
        i = smallint not null
begin
        i = 1;
end
```

The example below declares the simple local variable *total* and sets its initial value to 0. *Total* is not a parameter.

```
initialize () =
declare
        total = integer
begin
        total := 0;
end
```

The following example declares several local variables and hidden columns:

```
initialize () =
declare
  idnum = integer with null,
  idname = varchar(10),
  emptbl.empnum = smallint,
  emptbl.empsal = float with null,
  projtbl.projid = varchar(6)
begin
  idnum := NULL;
  idname := 'none';
end
```

In the example above, the current form contains table fields *emptbl* and *projtbl*. This initialization declares two local variables (*idnum* and *idname*) and initializes them with values. It also declares two hidden columns in the table field *emptbl* (*empnum* and *empsal*), and one hidden column in the table field *projtbl*. The local variables are set to the initial values "null" and "none."

Forms-Control Statements

A 4GL application involves the display and manipulation of data that the application user sees on forms in the window. Through forms-control statements, you can manipulate both form fields and local variables. You can use forms-control statements to:

- Position and move the cursor on the form
- Set the display mode of the form
- Clear the screen or clear individual fields
- Provide help text, display messages and prompts
- Validate data fields
- Manipulate individual table field rows or array records
- Process an entire table field or array at once
- Display additional operations on submenus

You can include forms-control statements as part of initialization or place them throughout the 4GL specification.

Positioning the Cursor

The resume statement and its variations control the placement of the cursor on a displayed form.

The resume statement returns the cursor to the current field. This statement is especially useful when the frame is first called and as the cursor returns to the screen following an activation.

The resume statement breaks out of the current operation before moving the cursor to the indicated place. Thus, you can use resume to break out of loops. Because resume closes the current operation, it is usually the last statement in an operation specification. However, you can place the resume anywhere in a 4GL specification.

If the form has no fields or if all fields are "display only," the cursor rests on the menu line. The cursor can rest on a table field in read mode.

Cursor position has these limitations:

- The cursor cannot rest on a derived field.
- The cursor can never rest on a field designated "display only," regardless of its mode.
- The cursor cannot rest on a field designated "query only," except when the form is in query or fill mode.
- The cursor cannot rest on a "display only" simple field. However, it can rest on the first column of a display-only table field, all of whose columns are display only.

Initial Cursor Position

As a frame starts up, the default positioning of the cursor is on the field with sequence number 1 as defined in VIFRED through the ABF FormEdit operation.

- If all fields are "display only" or "query only," or if the form has no fields, the cursor starts on the menu line.
- If you want to position the cursor somewhere other than at its default starting point, include a resume field statement as the last statement in the initialization.

The example below places the cursor on the Empname field, whether or not Empname has sequence number 1:

```
initialize =
begin
  resume field empname;
end
```

Repositioning the Cursor

Whenever control returns to the application user after an activation, the cursor returns by default to the field on which it was resting before the activation. You can position the cursor elsewhere by using one of these variants of the resume statement:

resume entry

Leaves the cursor on the current field and causes any before activation for that field to occur

resume field

Positions the cursor on a field or column on the form (except for derived and display-only fields) and causes any entry activation set for that field to occur

resume menu

Positions the cursor on the menu line

resume next

Must appear within a field activation. It completes the operation that caused the field activation to take place, which can be a cursor movement, menu, or key activation.

resume nextfield

Must appear within an initialization or activation. The resume nextfield statement positions the cursor on the next accessible field on the form, and causes any before activation for that field to occur.

resume previousfield

Must appear within an initialization or activation. The resume previousfield statement positions the cursor on the previous field on the form that is accessible, and causes any before activation for that field to occur.

Display Modes

Display modes affect field behavior, such as the ability to accept data. Form display mode affects the behavior of simple fields. Table field mode is controlled independently of the form mode. These 4GL statements control form modes:

```
set_forms form (mode = mode)
mode
```

These 4GL statements control table field modes:

```
inittable
set_forms field formname (mode(tablefieldname) =
    modetype)
```

Form Display Modes

The following are the four basic modes for the form as a whole, controlled by the set_forms statement:

fill

Default mode. Clears all visible fields except for default values specified when the form is edited with VIFRED. In fill mode, the user can enter or change data. Entry and exit activations occur.

Validations, if specified in VIFRED, are performed subject to set_forms frs settings.

update

Allows the user to enter or change data but does not clear fields before initialization. Entry and exit activations occur.

Validations, if specified in VIFRED, are performed subject to set_forms frs settings.

read

Allows the user to look at but not change the data on the form. Certain letters, when typed in this mode, behave as if typed in conjunction with the Control key (for example, p produces CTRL-P). Exit activations do not occur if the form is in read mode.

Validations are not performed.

query

Presents a blank form in which you can enter data to build a query on the database. This mode allows you to enter comparison operators (=, >, <=, and so on) as well as data. A form must be in query mode to allow the use of the qualification function in a database query. Entry activations occur if the form is in query mode.

Only data type checking, not full validation checking, is performed.

By default, a newly displayed form is in fill mode. To change this, use the set_forms statement. In this example, the statement occurs in an initialize statement:

```
initialize =
begin
  set_forms form (mode = 'query');
end
```

Use the set_forms statement in preference to the mode statement when you only want to change the mode. The mode statement is equivalent to reinitializing the frame.

Display Modes for Table Fields

4GL allows you to set table-field modes independently of the form as a whole, with this exception: When the form is in read mode, table fields are also forced to behave as if they are in read mode. When the form is set to some other mode, the table field's true mode behavior is restored.

The four basic modes for table fields are as follows:

fill

Default mode. You can change or add to the displayed data by opening new rows. Validations, if specified in VIFRED, are performed subject to set_forms frs settings.

update

You can change displayed data but not add to it. Validations, if specified in VIFRED, are performed subject to set_forms frs settings.

read

You can look at the data but not modify it or open new rows. Validations are not performed.

query

Allows you to change or add to the displayed data. Can use query operators. Only data type checking, not full validation checking, is performed.

The default display mode for table fields is fill. The inittable statement changes a table field's display mode and clears any associated data. The example below clears the table field "emptbl" and resets it to update mode:

```
inittable emptbl update;
```

Inittable erases any values a table field receives from another frame through the parameter list of a callframe statement. Do not use inittable in the initialize block in this case. Instead, place the query that loads the table field in the current frame's initialize block, after inittable. For example:

```
inittable emptbl read;
emptbl = select * from emp;
```

Clearing a Screen or Form

To erase one or more fields on the currently displayed form, or to blank the entire screen, use these statements:

```
clear field fieldname [, fieldname ]
clear field all
clear screen
```

Clear field initializes fields to blank, zero, or null values depending on the nullability of each field. It does not operate directly on derived fields. A derived field is cleared indirectly, however, when one of its source fields is cleared. This example clears the Name and Age fields on the current form:

```
clear field name, age;
```

Note: Mode *update* restores previous field values, even if the fields have been cleared. To avoid restoring previous values, use set forms form (mode = *update*).

Clear field all clears all displayed fields on a form. This does not include any local or global variables.

```
Clear field all;
```

Clear screen blanks the entire screen without clearing values from fields for the duration of the operation in which it is used. It then redraws the screen and redisplays the data. The following example clears the screen in preparation for displaying a prompt to the user:

```
initialize =

begin
        clear screen;
        answer = prompt 'Enter the correct code';
end
```

For more information about the clear statement, see clear Statement—Clears the Screen or Fields (see page 629).

Displaying Help Text Files

The following statements display a help text file within your application:

```
helpfile
help_forms
callproc help_field()
```

Typically, you write 4GL code so that the help file appears in response to a Help menu operation or to a function key such as FRS key 1. Use a text editor to enter the text for the help file.

The helpfile and help_forms statements and the help_field() built-in procedure provide application portability between Ingres environments.

- Use the helpfile statement to restrict access to the information placed in the named file (generally information about the current form). It does not access the Help utility. Use the helpfile statement to present help text if you want to restrict access to validation criteria or control/function key mappings.
- Use the help_forms statement to allow access to all aspects of the Help utility.
- Use the help_field() procedure to allow access to help on a particular field or column through the Field menu item of the help_forms statement. Help_field() displays a pop-up selection list of legal values or a field description.

The following example shows how to display a help file with the help_forms statement:

Windows:

```
'Help' =
begin
    help_forms (subject = 'Personnel frame',
    file = c:'\usr\program\pers.hlp');
end;
```

UNIX:

```
'Help' =
begin
    help_forms (subject = 'Personnel frame',
    file = '/usr/program/pers.hlp');
end;
```

VMS:

```
'Help' =
begin
    help_forms (subject = 'Personnel frame',
    file ='dra0:[usr.program]pers.hlp');
end;
```

If the specified file does not exist, FRS displays the message:

```
Sorry - cannot open help file on "Personnel frame"
```

For more information on these options for providing online help, see the sections on each in 4GL Statement Glossary (see page 935).

Displaying Messages and Prompts

Messages and prompts let you communicate with the user of the frame at run time. The statements are:

```
message message [with style = menuline | popup [option]]answer = prompt message [
with style = menuline | popup [option]]
```

The message statement displays a single or multi-line text string in the window, either on the menu line or in a pop-up box, depending on the message style you specify.

For menu-line display, either use the message statement alone, as in the following examples, or use the optional clause: with style= menuline. A convenient use of the message statement is immediately before a query statement:

```
message 'Retrieving employee data';
query statement;
```

A menu-line style message followed by a query statement stays on the frame until control is returned to the user. To control the length of time the menu line message remains in the window, use a sleep statement along with the menu-line message statement, as in this example:

```
message 'That field must contain a number';
sleep 5;
```

Use pop-up style if you want to ensure that the user sees a message. The instruction "Press Return" is automatically appended to a message displayed in a pop-up box. The message stays in the window until the user presses Return.

A prompt statement requests input from the user. The prompt displays a character string in the window, then accepts user input and assigns it to the specified field or variable, as shown in the example below:

```
repname = prompt 'Enter the report name: ';
```

When the user responds to this prompt at run time, the specified report name is assigned to the Repname field.

You can also use a prompt when you want to make sure the user sees a message. The user must respond to a prompt with a keystroke before the program proceeds. Tell the user to press the Return key here if no other response is needed.

Pop-up Messages

Use the message statement with the with style clause to display messages in a pop-up box. The message appears in a box on top of the currently displayed frame. This example displays a message in pop-up style at the current cursor position:

```
message :errorbuf
  with style = popup (floating);
```

The rest of the frame remains intact. When the user presses Return, the pop-up message box disappears and the frame returns to its previous state.

You can position the pop-up box anywhere in the window, depending on the size of the pop-up. For example, the following statement causes the prompt (in quotes) to appear in a pop-up box in the upper left-hand area of the frame. The number coordinates place the top left hand corner of the box in the second column and the eighth row.

```
Message 'An error has occurred'
  with style = popup (startcolumn = 2,
  startrow = 8);
```

To specify the dimensions of the message box, use a statement like this:

```
message 'An error has occurred'
  with style = popup (rows = 12,
  columns = 25);
```

This statement produces a box that is 12 rows in height and 25 columns in width.

Validating Data Fields

Data in fields is validated according to criteria established in VIFRED when you create the form using the ABF FormEdit operation. Most validations are triggered by cursor movement out of fields. The three phases of the validation process check that:

- A required (mandatory) field has an entry
- The field has the correct data type
- Any "custom" validation check specified in the VIFRED form definition is performed

The statements for explicit validation of existing data are:

```
validate
validrow
validate field
```

The validate statement checks every field in the current form for which validation checks were specified in VIFRED through the ABF FormEdit operation.

The validate field statement performs validation checks on simple fields or table fields as you have defined them in VIFRED. Validating rows in table fields is covered in Manipulating Table Fields and Arrays (see page 865).

If a field fails a check, FRS displays an error message and places the cursor in the field that failed so that the user can enter correct data. For example, if you have specified a validation check of ">=20000" for the *salary* field, the following statement checks the value entered by the user:

```
validate field salary;
```

If the amount entered is less than 20,000, the field fails the validation check.

While you cannot perform a validation on a derived field directly, any source fields must be valid in order for the value of the source field to be defined.

Validation and Mode

The display mode of the form affects whether validation checks are made on the simple fields or table fields. The following table summarizes these effects. In the table:

Y indicates that a validation check takes place
N indicates that a check does not take place
D indicates a data type check only

Footnotes to entries in the Action or Statement column follow these tables.

Simple Fields	**Forms Display Mode**			
Action or Statement	**Fill**	**Read**	**Update**	**Query**
Field validated on leaving field? (1)	Y	N	Y	N
Form field validated when program references it?	Y	Y	Y	D
Form field validated when program assigns a value?	D	D	D	D
Field validated when validate field statement executes?	Y	Y	Y	D
Fields validated when return statement executes? (2)	N	N	N	N

Table Fields	**Table Field Display Mode**			
Action or Statement	**Fill**	**Read**	**Update**	**Query**
Field validated on leaving column or table field? (1)	Y	N	Y	D
Field validated on tabbing backward to previous column or out of table field? (1)	N	N	N	N
Current row validated on moving to row above or below? (3)	Y	N	Y	D
Column validated when program references a value in the table-field column?	Y	Y	Y	D
Column validated when program assigns a value to the table-field column?	D	D	D	D
Row validated on validrow statement?	Y	Y	Y	D
Visible rows validated before rows in table field scroll out of view, due to user action or program scroll statement?	Y	N	Y	D

Table Fields	**Table Field Display Mode**			
Action or Statement	**Fill**	**Read**	**Update**	**Query**
Loaded rows validated before loadtable statement executes?	N	N	N	N
Visible rows validated before unloadtable or deleterow statement executes? (4)	Y	N	Y	D
Rows *not* visible validated before unloadtable statement executes?	N	N	N	N
Validate row to be deleted before deleterow statement executes?	N	N	N	N
Visible rows validated before successful deleterow or insertrow statement?	Y	N	Y	D

Key Validations	**Form Display Mode**			
Action or Statement	**Fill**	**Read**	**Update**	**Query**
Validation on pressing Menu key or mapped FRS key with validation checking on (through validate=1 on menu line or set_forms/set_frs)?	Y	Y	Y	D
Validation on pressing Menu key or mapped FRS key with validation checking not on? (1)	N	N	N	N

Footnotes to previous tables:

1. Default validations. You can turn this validation off or on with a set_forms frs validate statement.
2. Except displayed fields whose values are supplied by a byref statement. These are validated before the called frame returns.
3. Cannot override these with the set_forms frs validate statement.
4. If a validation check fails, the unloadtable loop ends and no rows are returned.

Row State and Record Values for Table Fields and Arrays

Each record in a table-field data set or array has an associated integer state value. You can reference the special constant _state to obtain this value for each record, or assign _state to a variable in an unloadtable statement. In addition, you can reference the special value _record to get the record's sequence number in the table field data set or array.

In a table field, _state indicates whether data in a data set record is Undefined, New, Unchanged, Changed, or Deleted. A record becomes Changed when the user types over it, or when its value is changed by a statement in the application.

By default, table-field _state and _record values are defined implicitly and maintained automatically by the Forms Runtime System (FRS). You can override the default value of Unchanged by assigning a value of Undefined, New, or Changed to _state when using insertrow to insert a row into a table field or a record into an array.

In an array, _state is primarily under programmer control. By default, _state is set to Unchanged when an array record is created and to Deleted when it is deleted. However, if a record is created through insertrow, you can specify a state of New or Changed, or Unchanged. This is useful primarily for applications in which you want to use an array that is based on a table field. You cannot set _state through an assignment statement.

In an array, _record is always set to the 1-relative position of the record in the array, and cannot be explicitly set. For deleted records, _record is negative or zero. Note that arrays do not have undefined rows.

You can set the _state of table fields and arrays using the insertrow and deleterow statements. These statements are discussed in Manipulating Table Fields and Arrays (see page 865).

The following table provides default values of _state for a table field row and an array record.

State	Value	Description
Undefined	0	Empty table-field row appended but not filled by the application user.
New	1	Table-field row appended and filled by the user at run time.
Unchanged	2	Table-field row loaded or inserted by a 4GL statement, not changed by user. Row state after insertrow is (by default) 2.

State	Value	Description
Changed	3	Table-field row inserted by a 4GL statement but modified since by the user or by another 4GL statement.
Deleted	4	Table-field row inserted by a 4GL statement with state set to "Changed" or "Unchanged" but since deleted by another statement.
New	1	Record created by insertrow with _state = 1. Record has not been deleted.
Unchanged	2	Record created with _state unspecified *or* _state = 2. Record has not been deleted.
Changed	3	Record created by insertrow with _state = 3. Record has not been deleted.
Deleted	4	Array record marked for deletion by the deleterow statement or the arraysetrowdeleted procedure.

Manipulating Table Fields and Arrays

4GL provides the following statements for manipulating rows and columns in table fields:

- validrow
- insertrow
- clearrow
- deleterow
- loadtable

The first four of these statements operate on individual rows of the table field's visible display. The loadtable statement operates on the table field's dataset; this can affect the visual display.

To manipulate all rows in a table field's data set, use the unloadtable statement. This statement sets up a loop that processes all the rows sequentially. You cannot indicate specific rows in an unloadtable loop.

The insertrow, clearrow, and deleterow statements also operate on the records in arrays. In addition, there are a number of built-in 4GL procedures that you can use to operate on arrays. See Built-In Frames and Procedures for a brief description of these procedures.

Records and Rows in Table Fields

A table field consists of two parts:

- An underlying data set that contains all records retrieved from the database
- A display window that contains rows visible to the user; this generally is a portion of the dataset

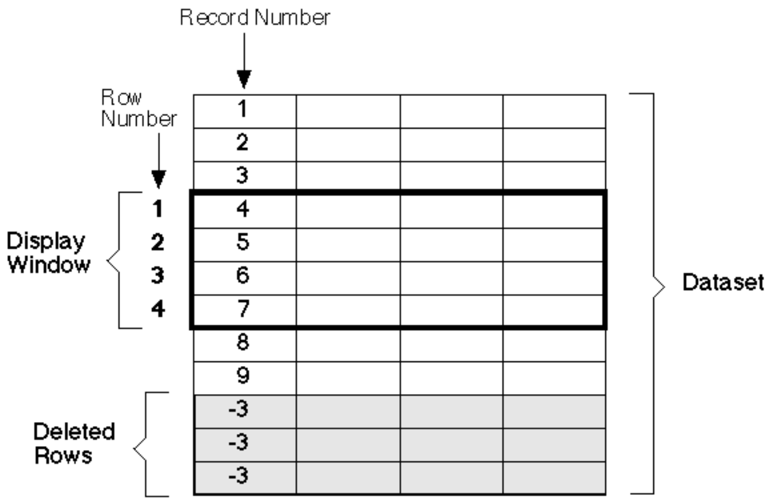
Except for the loadtable and unloadtable statements, all the statements that you use with table fields operate only on rows in the visible display window. For example, the insertrow statement inserts a row after the row number that you specify. If this row currently is not visible, you can use the scroll statement to move it into the display window.

The position of a record in the data set (as determined by the _record constant) can be different from its position in the visible display. When you use a table field statement such as insertrow, you always specify the row number in the display window.

The following figure illustrates the relationship between a table field's data set and display window. Deleted rows are handled as follows:

- Deleted rows whose state previously was "Changed" or "Unchanged" are moved to the end of the data set (that is, the unloadtable statement processes them after all other records). The rows are given non-positive record numbers.
- Deleted rows whose state previously was "Undefined" or "New" are removed from the dataset altogether.

You cannot Mode scroll to a deleted row.



Validating a Row

Use the validrow statement to perform a validation check on columns in a particular row of the table field. Ingres validates the indicated columns in the row on which the cursor is currently resting, unless you specify a row number, as in the statement below:

```
validrow partstbl[3] (partno, cost);
```

To validate all the table field's displayed rows at one time, use validate field *tablefieldname*. The validate and validrow statements perform validation checks on columns as you have defined them with the ABF FormEdit operation. If any field fails a check, ABF displays an error message and places the cursor in the field that failed, to permit reentry of data. The validrow statement does not work with arrays, as arrays do not have validations.

Inserting a Row or Record

The insertrow statement opens up a new row in a table field or a new record in an array following the row or record you specify by number.

When adding rows to a table field, you are dealing only with visible rows in the window. If you do not specify a numbered row, the statement inserts a row just after the row on which the cursor rests. If you enter 0, the statement inserts a new top row in the display. The following statement opens up a new row immediately following row 4 in the table field *partstbl*.

```
insertrow partstbl[4]
```

This is the fourth visible row in the window; the number has no relationship to the position the row might have in the underlying data set of the table field. Assuming that the table field's mode is fill, update, or query, the user can enter or change values in the new row.

When adding records to an array, the insertrow statement inserts a new record after the record designated by the number within the square brackets. You must include such a number when adding a record to an array. This number refers only to the location of the record in the array, and has no relationship to the number the record might have if displayed in the window in a table field.

If you enter 0, insertrow inserts a record at the beginning of the array. (Remember that the first record in an array is numbered 1.) The default _state attribute of the new record is "Unchanged." You can specify a different value for _state in the insertrow statement.

To add new records to an array, either use the insertrow statement or reference the record that is one position beyond the last record in the dataset.

You also can use the arrayinsertrow() built-in function to add records to an array. In this case, the record is added at the index number that you specify, and all the following records are renumbered. See the section ArrayInsertRow() for more information on using this function.

For example, to load an array with constant values, specify several insertrow statements in their 4GL code:

```
insertrow array[0]
  (coll = 1.0, col2 = 'Record1');
insertrow array[1]
  (coll = 2.0, col2 = 'Record2');
insertrow array[2]
  (coll = 3.0, col2 = 'Record3');
```

You also can insert new records by direct reference:

```
array[1].coll = 1.0;
array[1].col2 = 'Record1';
array[2].col2 = 2.0;
array[2].col2 = 'Record2';
array[3].col3 = 3.0;
array[3].col2 = 'Record3';
```

For more information, see insertrow Statement—Insert a Row (see page 692).

You can use the loadtable statement to append a row after the last row of a table field's dataset, eliminating concern about which rows of the table field currently are displayed.

The loadtable statement has the following syntax:

```
loadtable tablefieldname (columnname = expression
    {,columnname = expression} )
```

You cannot use the loadtable statement with arrays.

Clearing or Deleting a Row or Record

To clear a row or specific columns within a row of a table field, use the clearrow statement. The FRS clears the row, leaving a blank row. The cursor remains on this row unless you specify a different (displayed) row number. If you specify column names as well, the FRS clears only those columns.

The following example clears the *cost* and *total* columns of the row on which the cursor is resting in the currently displayed table field *partstbl*:

```
clearrow partstbl (cost, total);
```

The deleterow statement deletes a record from a table field or array and closes up the table or array after the deletion. When you use the deleterow statement on a record of an array, the record's _state value is set to "Deleted" and its record number becomes non-positive. Records to be deleted are assigned record numbers beginning with "0," then "-1," and so on.

For example, assume you have an array called "Emparray" that contains two deleted records and four non-deleted records. The following statement:

```
deleterow emparray[3]
```

deletes the third non-deleted record. This record has its _state value set to "4" and its _record value set to "-2." You now can refer to this record explicitly as "emparray[-2]."

When you use the deleterow statement on a row of a table field, one of the following situations occurs:

- If the row's _state value was "0" (Undefined) or "-1" (New), then the row is removed from both the table field display window and the table field's dataset.
- If the row's _state value was "2" (Unchanged) or "3" (Changed), then the row is removed from the table field's display window, but it remains in the table field's data set.

  However, its _state value is set to "4" (Deleted), and its _record value is set to "-3." The deleted record is accessible only through an unloadtable loop.

  For example, assume the table field "partstbl" currently displays the fifth record of the underlying dataset in the third row of the display window. The current state of this record is "2" (Unchanged). Then the statement:

  ```
  deleterow partstbl[3]
  ```

  deletes from the display window the fifth record of the dataset. The record remains in the dataset, but its _record value is changed from "5" to "-3," and its _state value is changed from "2" (Unchanged) to "4" (Deleted).

Processing an Entire Table Field or Array with Unloadtable

The unloadtable statement executes a loop that unloads one row at a time from the data set of a table field or one record from an array, and performs any specified actions on the row or record. This allows the application to scan through a table field or array one row or record at a time, applying a series of 4GL statements to each. Typical examples include:

- Query statement that uses the values in the table-field rows to update the rows in a database table
- Assignment from table-field rows to array records in copying a table to an array

The unloadtable statement executes a loop once for each row in the data set. The loop continues until either all rows are processed or an endloop or resume statement is encountered.

Enclose the statement list for the unloadtable statement in braces or the keywords begin and end.

Using Unloadtable with Database Access

You can use the unloadtable statement to update a database table from a table field that is in fill or update mode.

For more information on Forms Runtime operations, see the appendixes in *Character-based Querying and Reporting Tools User Guide.*

The application in the following example uses the _state constant and an unloadtable statement to update a database with mailing list information.

```
'Writelist' =
begin
  unloadtable maillist
  begin
    if :maillist._state = 1 then
      /* Add row to database */
      insert into mailtable
        (name, address)
        values (maillist.name,
          maillist.address );
    elseif :maillist._state = 3
      /* Update the row in the database */
      update mailtable
        set name = :maillist.name,
        address = :maillist.address
        where name = :maillist.old;
    elseif :maillist._state = 4 then
      /* Delete the row from the database */
      delete from mailtable
        where name = :maillist.old;
    endif;
  end;
  commit;
end
```

When this application is running, the user can choose Writelist from the menu to make changes to the database. Ingres, depending on the value of the _state constant, performs the appropriate database operation.

The key for the update and delete statements in this example is the table field's hidden column *old*, which contains the value of the database column *name* that was originally loaded from the database (before any updating or deleting by the application user). Note that the updated row is identified by the original value still in the hidden column *old*.

You must declare the hidden column in the initialization section for the frame and load the values into it along with those in the visible columns when the database retrieval occurs.

Row Processing Order

The unloadtable statement processes deleted records last. It processes the deleted records in the reverse order in which they were deleted (that is, beginning with the most recently deleted record). To process any deleted records either first or in the order in which they originally appeared, use the following procedures:

- To process deleted records first, unload the table field or array twice. The first time, process only deleted records. The second time, process all other records.
- To process deleted records in the order they originally appeared in the table field, do not use the deleterow statement to delete records. Instead, mark deleted records visually so that the user can see they are deleted. In addition, mark them internally with a hidden column or array attribute so your program can see they are deleted in the unloadtable loop.

For more information, see unloadtable Statement—Loop Through Rows and Execute Statements (see page 748).

The Run Submenu and Display Submenu Statements

4GL supports two statements, run submenu and display submenu, which allow you to display a submenu without an attached query. The submenu is a display construct consisting of the double keywords run submenu or display submenu followed by an operation list consisting of activations and statement blocks. It replaces the original menu line for the form.

The principle difference between the run submenu and display submenu statements is in the way each is used:

- Use display submenu with an operation list when you want the form to remain active; the user has access to the fields on the form while the menu is displayed.
- Use run submenu with an operation list if you want the form to be inactive; the cursor stays on the menu line, and you must make a selection before regaining access to the form.

The following example shows a display submenu block, which lets you insert, delete, and clear rows while updating a table field.

```
'Employee_table' =
begin
  display submenu
  begin

    'Insert' =
    begin
        insertrow emptbl;
    end

    'Delete' =
    begin
        deleterow emptbl;
    end

    'End', key frskey3 =
    begin
        endloop;
    end

  end
end
```

These two statements are discussed in more detail in 4GL Statement Glossary (see page 935).

Database Access Statements

Database access is an important part of forms applications. Applications display data retrieved from the database on the form and, for multiple rows, allow the user to scroll through the rows of data, displaying each row in the window.

4GL database access statements provide virtually the full power of the Ingres query language. In most cases, there is no difference between the syntax of a query language statement and the statement's corresponding 4GL version. In 4GL, you use these SQL statements to query the database:

- select
- insert
- update
- delete

These statements enable you to retrieve data, append rows to tables, update rows in tables, and delete information that is no longer needed.

The following section introduces these statements. For more information, see the appropriate section in 4GL Statement Glossary (see page 935), and the *SQL Reference Guide.*

Using the Select Statement

To read data from database tables into forms, fields, and variables, use the select statement. The select statement takes values from the columns in the specified database table for all rows that satisfy the condition specified in an optional where clause. For example, the following example returns information about the employee whose employee number is 7 from the Personnel database table to the form Deptform:

```
deptform := select lname, fname
  from personnel
  where empnum = 7;
```

Variants of the select statement allow you to read data into a variety of query targets, display a submenu to use with the results of the query, or display the rows retrieved by a query for processing one at a time. For a detailed discussion of this statement, see the Select section.

Using Insert to Add Data to Database Tables

To insert new rows of data into a database table, use the insert statement. For example, this statement adds a row to the Personnel table, transferring the contents of simple fields Lname, Fname, and Empnum to table columns *last*, *first*, and *empnum*, respectively:

```
insert into personnel (last, first, empnum)
  values (lname, fname, empnum);
```

Appending values to a database table from a table field generally involves using the insert statement along with the unloadtable statement, which is described in Processing an Entire Table Field or Array with Unloadtable (see page 870) and unloadtable Statement—Loop Through Rows and Execute Statements (see page 748). Unloadtable can also be used with arrays.

Using Update to Change a Database Row

The update statement lets you update the values of columns in a database table using values from fields or simple variables, based on the conditions specified in the where clause. The values can come from simple fields, table field columns, or local or global variables. For example, the following statement updates any row in the *personnel* table with a value in the *idnum* column equal to the value in the *idnum* field on the current form.

```
update personnel
  set salary = :sal, manager = :manager
  where idnum = :idnum;
```

In the update statement, prefix field names in the current form with a colon to eliminate any possible conflict between the names of fields and database columns. For additional information about runtime substitution in expressions and 4GL names, see "."

Updating a database table from a table field generally involves using the update statement along with the unloadtable statement, as described in Processing an Entire Table Field or Array with Unloadtable (see page 870) and unloadtable Statement—Loop Through Rows and Execute Statements (see page 748).

Using the Delete Statement

The delete statement removes rows from a database table, based on the conditions specified in the where clause. If no condition is specified, the statement deletes all the rows in the table. The statement below deletes any row of the Personnel table that contains the same value in the *empno* column as in the form's field Empno.

```
delete from personnel where empno = :empno;
```

To remove values from a database table using a table field in this way, use the delete statement along with the unloadtable statement, as described in Processing an Entire Table Field or Array with Unloadtable (see page 870) and unloadtable Statement—Loop Through Rows and Execute Statements (see page 748).

Committing the Transaction

SQL transaction processing is fully supported in ABF/4GL.

A transaction is opened on the first occurrence of a query statement (select, insert, update or delete) and remains open until you specifically commit it with the commit statement. You can use the following transaction control statements:

commit

Ends a transaction block, and commits the results of the transaction to the database

rollback

Terminates a transaction in progress, undoing the effects of all processed statements

savepoint

Declares a savepoint. You can roll a transaction back to any of its savepoints prior to committing. (For gateways users, the transaction is rolled back to the beginning.)

set autocommit on

Changes every query statement to a "Single Query" transaction with an implicit commit occurring after every successful statement

See the *SQL Reference Guide* for detailed descriptions of these commands and a full description of transactions in SQL.

In your 4GL programs, issue a commit statement after every query statement or group of query statements that form a single transaction. If you do not use explicit commit statements after queries (or begin/end transaction statements around queries) and do not specify set autocommit on, your 4GL/SQL application runs as a single transaction.

Each query issued accumulates as part of the transaction; this transaction does not commit until the user exits the application.

This large transaction can potentially acquire and hold locks on large portions of the database, drastically lowering concurrency and using up a large portion of the server's logging file. Statements within your 4GL code are not the only cause for locking in the database. The first time a frame is called, ABF prepares the form for use. If the form uses a VIFRED validation of the type "field in table.column," ABF issues a database query to retrieve validation data and loads the data into memory.

- If no transaction is open before the frame is called, ABF issues a commit statement, which releases the read locks on the validation data tables.
- However, if a transaction is open when the frame is called, a commit is not issued and the read locks on the validation table remain until your application issues a commit.

For this reason, it is good practice to commit any open transactions before calling a new frame.

Multi-Query Transaction Example

When multiple queries are involved in a transaction, both queries must run successfully or neither query runs. If a deadlock occurs, the transaction must rerun.

In the example below, the queries change the value of a field used to join two tables. If an error occurs, the transaction is rolled back and the join fields in both tables retain their original values. If the error is deadlocked, the example retries the transaction. The updates are committed only when both queries run successfully.

The example uses two integer global constants, OK and Fail. OK is defined to a value of 1 (one) and Fail is defined to a value of 0 (zero).

```
/* The example assumes SQL transaction */
/* semantics: set autocommmit off */
status = ok;
  /* ok/fail = global constants */
commit;
a: while (1=1) do
  b: while (1=1) do
  update orders set order_no = :new_order_no
    where order_no = :old_order_no;
  inquire_sql (errno = errorno,
    rows = rowcount);
  if (errno = 4700) or (errno = 4706) then
  /* deadlock or forced abort limit */
  /* reached; restart */
    endloop b;
  elseif (errno > 0 or rows = 0) then
  /* error: roll back transaction */
    rollback;
    status = fail;
    endloop a;
  endif;

  update order_items set order_no =
    :new_order_no
    where order_no = :old_order_no;
  inquire_sql (errno = errorno,
    rows = rowcount);
  if (errno = 4700) or (errno = 4706) then
  /* deadlock or forced abort limit */
  /* reached; restart */
    endloop b;
  elseif (errno > 0 or rows = 0) then
  /* error: roll back transaction */
    rollback;
    status = fail;
    endloop a;
  endif;

  /* successful transactions pass through */
  /* here */
  commit;
  endloop a;
  endwhile;   /* end of loop b */
  /* deadlocks pass through here; */
  /* outer loop reruns */
endwhile;   /* end of loop a */
if (status = ok) then
  /* success above */
else
  /* error above */
endif;
```

Repeated Queries

4GL provides an optimizing repeated version of the SQL database query statements: select, insert, update and delete.

The syntax for repeated queries varies from the standard data manipulation commands only in that the word repeated precedes the statement:

```
[repeated] select|insert|update|delete statement
```

Repeated queries run faster the second and subsequent times they are issued, compared to a query coded without the repeat feature. This can be a major benefit for applications in which the same query is executed repeatedly over the course of the user's session, varying only in that the values supplied to the fields or items in the where clause differ from one execution to another.

On the first execution of a repeated query, Ingres stores an optimized execution plan for the query. On subsequent executions, the work of parsing and optimizing the query need not be redone. The first execution of a repeated query is slightly slower than a nonrepeated query because of the work required to store the query's execution plan. On subsequent executions, the query runs significantly faster than a nonrepeated query, because the parse and optimize steps are skipped.

Repeated queries require additional memory overhead in the database server to store the query execution plan. That memory stays allocated for as long as the current session remains active. However, because all application users share the memory, multiple users can optimize resources.

Repeated statements are often implemented in a procedure or in a loop, where the same query can be executed many times. Because the repeated query is optimized separately as it reappears in different places in the 4GL code, it is important to modularize repeated query code.

The best candidates for repeated queries are queries run three or more times during the life of the application. Queries run inside an unloadtable loop and those in a submenu are often good candidates.

Restrictions

You cannot repeat the following types of queries:

- Queries that use the 4GL qualification function. For example:

  ```
  /* incorrect */
  repeated delete from employees where
    qualification (...);
  ```

- Queries that specify an entire where clause as a variable. For example:

  ```
  /* incorrect */
  repeated delete from employees where :w;
  ```

- Queries that use database table names or column names as variables. For example:

  ```
  /* incorrect */
  repeated delete from :t where
    last_name = 'Smith';
  ```

- Queries that specify the sort order of a column ("asc" or "desc") as a variable. For example:

  ```
  /* incorrect */
  repeated select * from employees
  order by last_name :direction
  ```

If you specify any of the above types of queries in a 4GL application, the compiler issues a warning and ignores the repeated qualifier.

Examples

The following are examples of repeated queries.

This example selects all the fields on the form from the table where the value of *col* is greater than the value in *var*. Each time the select is run, a new value for *var* can be specified.

```
repeated select * from itable
  where col > :var;
```

The next example updates information on an employee. Each time the query is run, a new employee number is specified along with data currently on the form.

```
repeated update employees
  set * = :emptbl.all
  where empno = :empno;
```

The following example deletes a sample test result. Each time the query is run, a new value for sample number and test result code is used.

```
repeated delete results
  where sample_no = :s_no
  and res_code = :res_code;
```

The last example inserts data from the form into the database table *customers*.

```
repeated insert into customers (*)
  values (custform.all);
```

Minimizing the Number of Repeated Queries

You can minimize the number of repeated queries in an application by running them from 4GL procedures. Then any frame in the application that needs to run a query, such as inserting a row in a particular table, can call the 4GL procedure that does the insert and pass it the information from the new row.

The following examples demonstrate ways to minimize the number of repeated queries through the use of records and 4GL procedures. Several examples follow in which 4GL frame code calls 4GL procedures and passes them a record. These routines deal with the database table Emp, which is keyed unique on the Emp_id column.

Listings of the 4GL procedures, InsertEmp, SelectEmp, and UpdateEmp, follow the 4GL frame code which calls them. The examples use two integer global constants, OK and Fail. OK is defined to a value of 1 (one) and Fail is defined to a value of 0 (zero).

Insert a row: This example calls the 4GL procedure InsertEmp to add information about a new employee:

```
if (insertemp(r_emp = r_emp) != ok) then
  rollback;
  message 'Error inserting data ' +
    varchar(:r_emp.emp_id)
      with style = popup;
  resume;
else
  commit;
endif;    /* insert accomplished */
```

Select a row: This example calls the 4GL procedure SelectEmp to retrieve information about an employee:

```
/* r_emp = type of table emp */
if (selectemp(r_emp = r_emp, emp_id = id)
  != ok) then rollback;
  message 'Error selecting data ' +
    varchar(:id)
      with style = popup;
  resume;
else
  inquire_sql (rows = rowcount);
  commit; /* Resets errorno and rowcount */
  if rows = 0 then
    message 'Employee number '
      + varchar (:id) + ' not found.'
      with style = popup;
    resume;
  endif;
endif;    /*r_emp now contains selected data */
```

Update a row: This example calls the 4GL procedure UpdateEmp to update information about an employee:

```
if (updateemp(r_emp = r_emp,

    emp_id = id) != ok) then

  rollback;
  message 'Error updating data ' +
    varchar(:r_emp.emp_id)
      with style = popup;
  resume;
else
  commit;
endif;    /* update accomplished */
```

The 4GL procedures called in the code above follow:

```
/* Procedure InsertEmp */

/* Insert row into Emp table */
procedure insertemp (
  r_emp = type of table emp
  ) =
declare
  err = integer;
begin
  repeated insert into emp (emp_id, name,
    age, hire_date)
  values (r_emp.emp_id, r_emp.name,
    r_emp.age, r_emp.hire_date);
  inquire_sql (err = errorno);
  if (err != 0) then
    return fail;
  else
    return ok;
  endif;
end


/* Procedure SelectEmp */

/* Use unique key to select 1 row from */
/* the Emp table */
procedure selectemp (
  r_emp = type of table emp,
  emp_id = integer   /* unique key */
  ) =
declare
  err = integer;

begin
  r_emp = repeated select
    from emp
    where emp_id = :emp_id;
  inquire_sql (err = errorno);
  if (err != 0) then
    return fail;
  else
    return ok;
  endif;
end
```

```
/* Procedure UpdateEmp */

/* Use unique key to update 1 row */
/* from the Emp table */
procedure updateemp (
  r_emp = type of table emp,
  emp_id = integer   /* unique key */
  ) =
declare
  err = integer;
begin
  repeated update emp
  set emp_id = :r_emp.emp_id,
    name = :r_emp.name,
    age = :r_emp.age,
    hire_date = :r_emp.hire_date
  where emp_id = :emp_id;
  inquire_sql (err = errorno);
  if (err != 0) then
    return fail;
  else
    return ok;
  endif;
end
```

Using the Asterisk and All

The asterisk (*) is a wild card character that enables you to transfer values between the database and all the fields of a form in a single statement. This multiple assignment applies to simple fields and table-field columns, but it does not include local variables or hidden columns. If the asterisk is used, each of the displayed fields or table-field columns on the form must correspond in name and data type to a column in the database table.

The reserved word all matches simple fields in a form or columns in a table field with other form fields or database columns. The all syntax matches only *displayed* fields and columns; local variables and hidden columns do not participate in the matching. All is valid in the target or parameter lists of the following statements:

- callframe
- callproc
- insert
- update

The exact syntax for all varies considerably from statement to statement.

The select statement uses the "*" symbol to achieve similar results, as described below. As a rule, you use all with a form or table-field name, while "*" substitutes for the names of database columns.

Using the Asterisk with the Select Statement

This example demonstrates the use of the asterisk (*) with the select statement:

```
deptform := select * from personnel
  where empnum = 6;
```

Data is retrieved from the columns in the Personnel table into all the simple fields of the form Deptform.

You can use the asterisk with the **select** statement with table fields, forms, records, and arrays.

This example fetches data into all the columns of a table field, except for its hidden columns. The form name is optional.

```
deptform.tfield := select * from personnel;
```

This example declares an array of the type of the table Personnel and then uses the asterisk to populate it with all the data in the Personnel table:

```
arr = array of type of table personnel;
...
arr = select * from personnel;
```

For each simple field or table-field column in a form, there must be a database table column with the same name and a compatible data type. The database table can contain unmatched database columns, but the form cannot contain form fields or table-field columns that do not correspond to database columns.

Using the Asterisk and All in Update Statements

The multiple assignment is also possible with update statements. This statement updates a row in the *personnel* table from all the simple fields in *deptform*.

```
update personnel
  set * = :deptform.all
  where idnum = :idnum;
```

When you use all in an update statement, use the asterisk (*) in place of the list of database columns. You can use the asterisk and all with the update statement with table fields and forms. The asterisk and all are not used in update statements with records and arrays.

Using the Asterisk and All in Insert Statements

You can also use the asterisk (*) with the insert statement. The insert statement inserts rows into a database table from simple fields or table-field columns. The example statement adds a new row to the table called *part* from all the columns in row 3 of table field *partstbl*.

```
insert into part (*)

  values (partstbl[3].all);
```

The following statement inserts values from all the simple fields in the form Deptform into columns of the database table Personnel:

```
insert into personnel (*)
  values (deptform.all);
```

When you use this syntax, make sure that all the simple fields of Deptform match, by name and data type, columns in the Personnel table. The database table can contain unmatched columns, but there *cannot* be unmatched form fields. The 4GL compiler expands *deptform.*all into a values list based on the list of simple fields in the form. This variant of the insert statement requires the special symbol "*" as the column list.

You can also use the all syntax to insert values from a row in a table field into a database table. As with simple fields, each column in the table field must match a column in the database table.

The asterisk and all are not used in insert statements with records and arrays.

Flow-Control Statements

Flow-control statements control the processing flow in a frame or application by allowing you to establish conditions for performing specific statements.

In a sequential execution, 4GL performs the same sequence of actions every time the user selects a particular operation. In most applications, however, the data the user enters determines the actions to be performed. The actions taken are conditional with respect to the data, and are performed through flow-control statements. You can use comparison operators and arithmetic operators to describe a condition, then instruct Ingres to execute groups of flow-control statements if the condition is met.

These statements include:

- if-then (elseif, else, and endif)
- while (do and endwhile)
- endloop
- return
- exit

Using the If-Then-Else Statement Combination

The if-then-else statement combination is the simplest form of flow-control statement, letting you choose between alternative paths of execution. The if statement applies to logical (Boolean) expressions, which can include comparison operators and logical operators (AND, OR, and NOT). The example below shows how to control the flow of frames within an application on the basis of the *status* field on the current form:

```
if status = 'n' then
  callframe new;
elseif status = 'c' then
  callframe completed;
else
  callframe inprogress;
endif;
```

Using the While Statement

The while statement specifies a list of 4GL statements to be repeated while a given condition is true. The condition is tested at the start of each pass through the loop; if the condition is true, then the statements in the loop are executed. The endloop statement can be used to break out of the loop at any time, as shown in the next section.

The example below repeats a prompt five times or until a valid y (yes) or n (no) response is given:

```
answer := ' ';
requests := 0;
while (lowercase(left(answer,1)) != 'y'
  and lowercase(left(answer,1)) != 'n'
  and requests < 5) do
  answer := prompt 'Please answer Y or N: ';
  requests := requests + 1;
endwhile;
```

In this example, the prompt command places the user's response into the answer field.

The while statement can also set up a loop which terminates with an endloop or resume statement.

Using the Endloop Statement

Use the endloop statement to break out of various kinds of loops. You can use endloop to terminate processing that starts with a while statement or to break out of an unloadtable loop without completing the processing of every row in the data set. You can also use it to exit from a submenu and return to an application, as in this example:

```
'End' =
begin
  endloop;
end
```

You can also use the resume statement to break out of a loop and return the cursor to a specific position in the window.

Using the Return Statement

Use the return statement to return control in an application from a called frame or procedure to the frame or procedure that called it. You can pass a single value back to the calling frame at the same time. This value must match the data type of the field it is assigned to in the calling frame. You assign the called frame's return data type as you define the frame within ABF.

The example below returns a status code to the calling frame:

```
'End' =
begin
  return status;
end
```

You can use a procedure that returns a value either as an expression or in an expression in a 4GL statement. The data type of the returned value must be compatible with the requirements of the expression.

Using the Exit Statement

The **exit** statement closes an entire application and returns control to the level from which the user entered the application. For example:

```
'Quit' =
begin
  exit;
end
```

At least one menu operation Quit or Exit must be paired with an exit statement somewhere in the application as a way of returning to the operating system when the user has finished. A return statement from the first frame called has the same effect.

Call Statements

4GL **call** statements transfer control away from the current frame to any of five levels:

- Another frame
- A 4GL or 3GL procedure that can include embedded query language statements or a database procedure
- Another 4GL application
- An Ingres tool, such as QBF
- The operating system

The 4GL call statements are:

- callframe
- callproc
- call application
- call subsystem
- call system

Using the Callframe Statement

The callframe statement transfers control from one frame or 4GL procedure to another frame. The frame or 4GL procedure in which the statement appears is known as the *calling* frame or *procedure*, and the frame to which control is passed is known as the *called frame*. A user-specified, called frame can call another frame, and so on.

When the callframe statement executes, Ingres displays the new frame, and control passes to it. Typically, a called frame displays its own menu operations. The menu might include an End operation that issues a return statement, as described below. An example of the simplest type of callframe statement is:

```
callframe start;
```

This statement passes control to the frame named Start.

The ability to display new frames is one of a forms application's principal advantages over other types of applications. It is easy to use 4GL statements to sequence through frames in any order, using 4GL conditional statements. The user can determine, at run time, the order in which the frames appear, depending on the values of the data on the form. A particular frame can include operations that call any one of a variety of other frames. For examples, see Sample 4GL Application (see page 1187).

Using a Pop-up Frame

As with messages and prompts, you have the option of displaying frames in a pop-up style. Callframe's with style clause allows you to override the form's VIFRED style definition. For example:

```
callframe newframe with style = popup
  (startrow = floating) ;
```

The statement above displays the frame Newframe as a pop-up in floating position, that is, at the current cursor position (if there is room at this position).

You can also alter the full-screen display of a form, as in this example:

```
callframe newframe with style = fullscreen
  (screenwidth = narrow);
```

Here, the optional parameter *screenwidth* changes the fullscreen style to the narrower of the screen widths that you set up for the frame in VIFRED. See Callframe for more on screen width options.

Passing Parameters

When you call a procedure, a user-specified frame, or a report frame, you can pass values to it by means of a *parameter list*. This provides a way for frames to share information. The called frame can use the information received from the calling frame. More importantly, the calling frame can place the called frame in a particular state, so that the user sees continuity between frames. You cannot pass parameters to QBF or graphics frames.

Values for the parameter list can be literals, they can be supplied from the fields or variables of the calling frame, or they can be supplied by database queries. When you pass a parameter, its value transfers to a corresponding element inside the called frame or procedure. In the called frame, all fields, including local variables declared inside parentheses following the initialize keyword, can receive parameter values.

The example below calls the Newframe frame and places values in the fields on its form Newform:

```
callframe newframe
    (newform.name = 'Doe, John';
    newform.id = :empnum);
```

The value in the calling frame's field Empnum is thus passed to Newframe's field Id. Because the called form is not the currently active form, prefix the field names in the called frame with the name of the form.

In addition to the simple initializations of fields in the called frame described above, the callframe statement can also pass a query as an argument to the new frame. In this case, the query does more than simply initialize the fields; it starts a loop much like an attached query or submenu. The called frame can include next statements to display subsequent rows of the query, just as in the case of an attached query.

Passing queries between frames gives 4GL applications a great deal of power. This feature allows several different frames to call one common frame with different queries. The called frame offers the same operations for each of the different queries. You can pass only one query to the called frame.

To pass data from the database into a called frame, use a select statement to assign values to simple fields or a table field on the called frame's form, as in the example below:

```
callframe newframe (newform =
    select projnum, projname
      from projinfo
      where projnum = :projno);
```

In this example, the current frame calls Newframe, passing values from the Projinfo database table into a form named Newform. The where clause restricts the retrieval operation to a specific project number that appears in the current frame's Projno field.

Passing Complex Parameters

You cannot assign or easily copy data parameters such as arrays, records, tables, table fields, and forms. However, arrays and records can be passed to frames and 4GL procedures as keyword parameters. You can pass complex parameters to user-specified frames and 4GL procedures, but not to 3GL procedures, Report frames, and database procedures. As with simple data components in 4GL, you pass complex data components as parameters by name.

Unlike simple variables, this is implicitly by reference, because all access to complex parameters is by reference. Changes made to the complex parameters in the called frame or procedure are reflected in the variable in the calling frame or procedure. The byref() clause cannot be applied to complex variables.

The types of the parameters being passed and the receiving elements for complex data components must agree. You cannot pass records declared as type of form or type of tablefield to other forms or table fields, but only to other form- or table-based record types or arrays. You cannot pass a table field to another frame or procedure.

Passing Parameters to Report Frames

User frames written in 4GL can pass parameters to report frames. Parameters passed to fields on the "form for report" are displayed as initial values when the form is displayed.

If a parameter does not correspond to a field on the form, the parameter is simply passed on by the report frame when the report is run. Therefore, if no form or a form with no fields is associated with the report frame, all parameters passed to the report frame are passed directly to the report as parameters. For example:

```
dept_no := 2;
lname := 'Smith';
callframe reportframe (name = lname;
  deptno = dept_no);
```

If the report frame's form has the fields Name and Title, the Name field has the initial value "Smith" and the Title field is empty. If the user fills in "Manager" for the Title field and overwrites the Name field with "Blumberg," ABF issues the equivalent of the following operating system report writer call when the user runs the report:

```
report dbasename reportname
  (name = 'Blumberg',
  title = 'Manager', deptno = 2)
```

Returning a Value from Callframe

When the called 4GL frame returns control to the calling frame, it can use the return statement to pass back a single value. Among many possibilities, you can use this to send back a status code, which the calling frame can then use in conditional processing. The value returned can be of any Ingres data type, but you must declare the data type in the ABF Edit a USER Frame Definition window, or the Vision Moreinfo about a USER Frame window, for the called frame.

The calling frame can assign the returned value to a field or hidden field by using the following variant of the callframe statement:

fieldname | variable `:= callframe` ***newframe*** `... ;`

The ellipsis represents the argument list, if any. In this way, the calling frame can get information (a status value, for instance) back from the frame it just called. This information can affect the further operation of the calling frame. For more details, see Returning to the Top Frame (see page 1337) and Callframe (see page 965).

Passing Parameters by Value or by Reference

Simple fields, local variables, or table-field columns in a table-field row can be passed as parameters to a called frame or procedure by value or by reference.

- A parameter passed by value (the default) is not affected by any change made to the value by the called frame or procedure. The parameter can be any 4GL expression.

  For example, if an integer field containing a value of 2 is passed by value to a procedure, the field still has a value of 2 when control returns from the procedure, whether or not the value of the corresponding data element changed.

- A simple field, table-field column, or variable passed *by reference*, however, is affected by a change in the value of its corresponding parameter inside the called frame or procedure.

The argument can be either a simple field, a table-field column, or a simple local variable in the current frame. You can pass local variables declared in parentheses after the initialize keyword as well as those declared in the declare section. If the value in a field or variable is passed by reference and the value of its corresponding data element changes while the called frame or procedure is running, the field has the new value when control returns to the calling frame. This enables the calling frame to receive multiple values (in addition to the single return value).

To pass a simple field, table-field column, or variable by reference, you must precede it with the reserved word byref, as shown below:

```
byref (objectname)
```

Here, *objectname* is a field name, a table-field column name, or a variable name.

Records and arrays are always passed by reference; however, you must not use the byref() keyword with them.

The value for a field, table-field column, or variable passed by reference is not normally available to the calling frame or updated in the window of the calling frame until the current operation is complete. For example, if a menu operation passes a field by reference to a procedure and, on return from the procedure, continues with other statements, the field's value is not changed until all the statements in the menu operation are executed.

If you must change the value before the end of the operation, place a redisplay statement immediately after the callframe or callproc statement.

Using the Callproc Statement

The callproc statement calls a procedure from a frame or 4GL procedure in an application. A *procedure* is a series of statements written in 4GL, Ingres SQL (called a database procedure), or a 3GL (also called host language). You can include procedures anywhere in your application that you would use a frame (including the start frame.) When you call the procedure, Ingres executes the commands it contains.

You can pass parameters (in the form of expressions) to the procedure. A parameter list can include any 4GL expressions. You can pass parameters to a procedure by reference if you use the reserved word byref in front of the parameter name. You must specify, in the parameter list of the called procedure, the names of the parameters to which the values of the expressions are to be assigned.

Ingres checks at run time that any field or column that you pass to a procedure has a data type that is compatible with the parameter in the procedure. See the section Callproc for a complete description of callproc syntax.

Calling 3GL Procedures

An application can call a procedure written in a 3GL. A 3GL is a programming language such as C. These procedures generally include embedded query language statements.

For example, this 4GL statement calls the procedure named *compute*:

```
callproc compute;
```

The procedure name *compute* must be declared to ABF or Vision and its source code file must have the correct file extension. You declare a procedure with the Create a Procedure frame for the procedure type.

Just as you can pass arguments to frames, you can pass them to procedures, but the syntax is slightly different. 4GL requires the following syntax for calling a procedure with arguments:

```
callproc procedurename (argument {; argument});
```

The called procedure must be written to accept arguments in the same order in which the application passes them.

Parameters to 3GL procedures must be simple fields, simple variables, individual array attributes, record attributes, or table field cells; you can not pass records and arrays as parameters. Also, you cannot pass more than 40 parameters to a 3GL procedure.

Calling 4GL Procedures

An application also can call a procedure written in 4GL. This feature can help minimize the number of statements in your application in cases in which several frames or operations must perform an identical function. You can place the 4GL statements required to perform the common function into a 4GL procedure and call the procedure as required, with or without an argument list.

A 4GL procedure can be global or local to its source file. The following considerations apply:

- You must declare a global 4GL procedure on the ABF Create a Procedure frame, specifying the name of a source file that contains the procedure. The 4GL code that defines the procedure must appear at the beginning of the source file that you specify.
- A local 4GL procedure is a series of 4GL source statements that appear in the source file for a user-defined frame or a global procedure.

  You only can call local 4GL procedures from the source file in which they appear. The ABF Frame Catalog has no record of local procedures.

The syntax for calling a 4GL procedure with arguments is similar, but not identical, to that used in calling a programming language procedure:

```
callproc procedurename (parameter = argument
    {,parameter = argument});
```

The *parameter* is the name of the parameter in the called procedure to which the *argument* is being assigned. Its name must be the same as the name given it inside the procedure. Because of this restriction, you can pass the parameters in any order to the called procedure.

Every parameter when calling a 4GL procedure is not called. Any parameter in the called procedure that is not passed a value is given a default value according to its data type. However, any parameter specified in the callproc statement must exist as named in the called procedure. The parameter names are 4GL names.

The following simple example shows the basic pattern for writing 4GL procedures. Note that the data type of each argument must be declared in the procedure heading:

```
procedure addtax  (tax=float4,cost=float8) =
begin
  cost := cost + (cost * tax);
end
```

Call the procedure *addtax* with the following statement:

```
callproc addtax (cost = byref(costfield),
  tax = taxpercent);
```

This statement illustrates some of the issues discussed above. First, note that the order in which the parameters are specified in the callproc statement is not the same as the order in which they appear in the procedure's heading. They are, however, identical in name to the parameters within the procedure. Second, the procedure call passes the form field *costfield* as a way of extracting a value from the procedure.

When the operation that calls the procedure is completed, *costfield* is updated to reflect its new value. If this is a displayed field, the new value for the field also appears on the form.

You can use the reserved word all in the parameter list to a 4GL procedure, as shown below:

```
callproc procname (calling_formname.all |
    calling_tablefield_name[[integer_expr]].all)
```

This syntax maps the simple fields in the calling form (or columns in the named table field) to the parameter list inside the procedure. Any field or column that does not correspond to a parameter is ignored.

Local variables and hidden columns are not mapped. When **all** is used with a table-field name, values are taken from the row on which the cursor is currently resting. To override this, specify an integer expression enclosed by brackets following the table-field name. In this case values are taken from the row with that number.

Using a Procedure to Test Global Variables

In order for global variables in an application to work correctly, set the values of global variables before you execute any frames or procedures that use the variables.

You can create a procedure that uses an integer global variable in ABF to check whether another global variable in an application has been set and, if necessary, sets it. You can call this procedure from any frame that uses the second global variable.

The following example performs such a check and initialization:

```
if (G_init = 0) then
  G_sess_info := select dba = dbmsinfo('dba'),username =
    dbmsinfo('username');
  G_init = 1;
endif ;
```

The example uses an integer variable *G_init* to check whether the global variable *G_sess_info* has been initialized. The first time you call the procedure, *G_init* has the initial 4GL default value of 0. This causes the if clause to be executed.

G_sess_info is a global variable of a record type that has two attributes, *dba* and *username*. The procedure uses the 4GL dbmsinfo function to retrieve the current values for the DBA and application user into the attributes of *G_sess_info*.

The procedure then sets the value of *G_init* to 1, so that the select is not executed again unnecessarily the next time the procedure is called.

Calling Database Procedures

Database (SQL) procedures are library procedures. Call these as you do any library procedure. The following example calls the database procedure *addtax*:

```
totcost = callproc addtax
  (cost = costfield;
  tax = taxpercent);
```

For more information on database procedures, see the *SQL Reference Guide.*

Calling Ingres Applications

A variant of the call statement allows the user to start up a second Ingres application on the currently executing DBMS connection. The syntax for calling an application is an instance of the syntax for calling any Ingres tool. There are two versions:

```
call application (name = filename, frame = framename)
call application (executable = filename, frame =
  framename);
```

The value of *filename* can be either a full pathname or the name of a file in the current working directory. The *filename* and *framename* can be any expressions that evaluate to appropriate string constants.

The parameters executable and name are interchangeable.

VMS: The parameters executable and name have different syntax. The first syntax is:

```
call application (name = applicationname, frame = framename);
```

The *applicationname* is the name of an application in your current database, defined as a DCL command in your environment. The *framename* is the name of the first frame to be called in the application. These names can be any expressions that evaluate to appropriate string constants.

For example, the following statement starts up the application *inventory* beginning in the frame *parts*:

```
call application (name = 'inventory', frame = 'parts');
```

The second syntax is:

```
call application (executable = full_directory_specification,
 frame = framename);
```

The parameter executable must have as its value the full directory specification of the application's executable file. The full_directory_specification and framename can be any expressions that evaluate to appropriate string constants.

This statement uses the executable parameter:

```
call application (executable =
  'usr_disk:[devel.joe]myapp.exe',
  frame = 'start');
```

Using the Call Subsystem Statement

The call *subsystem* statement allows the user to call Ingres tools or other ABF or Vision applications directly from a running application. Parameters include the usual execution flags for Ingres tools. For example:

```
call qbf (table = 'employee');

call report (name = namefield,
  mode = modefield);
```

The first example calls the QBF interface. In the second example, Namefield and Modefield are fields in the current form.

You can call any of these Ingres tools from within an application. "Call System" provides the parameter list for each subsystem call.

- RBF/Report-Writer
- QBF
- VIFRED
- ABF
- Interactive SQL

Using the Call System Statement

Use the call system statement to call the operating system (and, optionally, to run a system program) from within an application. If you do not specify a command as a parameter in the call system statement, the application user enters the operating system level and sees the operating system prompt. If you specify a particular command, Ingres executes that command, then returns to the running application.

The example below calls the operating system without specifying a command:

```
call system;
```

This places the user at the operating system level. You can execute any command provided by the operating system. When you exit from the system, control returns to the current frame in the application.

The example below calls the system and executes the program called *expenses*, after which control returns to the current frame in the application:

```
call system 'expenses';
```

Do not use this call to start up a new Ingres application, because the Data Manager connection is started, which uses up system resources. Instead, use the call application statement where possible.

VMS: Do not use the VMS inquire command in a command file called with call system. For example the statement shown below does not execute properly:

```
$ inquire x "Enter data:  "   /* error */
```

The recommended alternative is as follows:

```
$ open/read userterm tt
$ read/prompt="Enter data: " userterm varname
$ ...
$ close userterm
```

Accessing Files from 4GL

You can use 4GL to access files at the operating system level, without leaving 4GL or calling a 3GL procedure. Use the following built-in functions to access files:

CloseFile()

Closes a file. To delete a file, use the closefile() statement with the clause disposition**=**'delete'.

FlushFile()

Writes the contents of a file to disk

InquireFile()

Retrieves information about a file

OpenFile()

Opens a file. To create a file, use the openfile() statement.

PositionFile()

Positions a file

ReadFile()

Reads a file

RewindFile()

Positions a file to 0

WriteFile()

Writes data to a file

The file-access functions are called with the callproc statement. The functions follow the syntax for the callproc statement, which is:

```
[returnvalue = ][callproc] function(param = value
    {, param = value})
```

A parameter can be passed as a string, an integer, a variable or a variable by reference. Parameters are case-insensitive.

The *returnvalue* is an integer that indicates the function's return status. A return value of 0 means the function completed without error. A return value of < 0 indicates the function failed. A runtime error displays a specific error message, including an operating system-specific error, if available.

For the complete syntax for each of these statements, see 4GL Statement Glossary (see page 935).

The file-access statements are not affected by database transaction handling. (For example, the rollback statement does not affect these statements.) These statements also cannot be called from a 3GL using the exec 4gl syntax.

File Handles

The handle is a file pointer that identifies a file opened with the openfile() statement. If you specify the handle by reference (with the keyword byref), 4GL assigns a negative integer and returns the value to you. If you do not specify byref, 4GL uses the value you provide, which must be a positive integer.

After you close a file, the 4GL-generated handle does not remain. If you specified a handle without byref, you can specify the same handle again for another opening of the file.

The 4GL file access statements require the handle to access the file. A file must have been previously opened with the openfile() statement and the handle must be known to 4GL before any of the other statements can be used. A handle cannot refer to more than one file.

A file must be on a local node to be opened with the openfile() statement.

No 4GL file locking is supported with these statements. See your operating system documentation for information on how your system handles file locking.

The number of files you can have open depends on your operating system and your system configuration. 4GL supports up to 2 gigabytes of handles.

File Types

A file can be opened as one of 3 types:

binary

A binary type file has fixed-length records and no newline markers. A binary file cannot be edited by the system editor. The record size can be specified when the file is opened. If the record size is not specified, the record size is determined from the first read or write. Each subsequent read or write (until the file is closed) must consist of records of the declared size. The record size is not stored when the file is closed and reopened.

text

A text type file can accept only string data types (char, varchar, c and text). A text file can be edited by the system editor if the file does not exceed the editor's line length or file size limits.

stream

A stream type file accepts any data in any order. The user must read the data in the same way it was written.

When you close and reopen a file, you must not change the type of the file. 4GL cannot read the data correctly if, for example, a file is created as a binary type and reopened as a text type file.

File Modes

A file can be opened in one of four modes. The mode determines where the file is positioned when opened, and what functions are allowed. After you close a file, you can reopen the file in a different mode, depending on what you are doing. For example, you can create a file in create mode. You can close the file and later reopen it in append mode to add more data. The mode can be:

read

Opens a file for reading of data. Read mode opens a file to the beginning. You cannot write to a file opened in read mode. You can reposition a binary or stream file in read mode.

update

Allows both reading and writing of data. Update mode opens a file to the beginning. You can reposition a binary or stream file in update mode. You can update a record by repositioning the file to the beginning of the record, then writing the new data.

create

Opens the specified file and positions it to the beginning. If a file by that name does not exist, the openfile() statement creates the file. If a file by that name exists, it overwrites the existing file, except in VMS.

VMS: 4GL creates a new version of the file. It does not overwrite the file unless the file version limit is reached.

To create a new file, open the file in create mode.

You cannot read, position or rewind a file opened with create mode.

append

Allows writing data to the end of a file. You cannot read, position or rewind a file opened with append mode.

Using File Modes and Types

All file types and modes can use the openfile(), closefile(), and inquirefile() statements. Other statements can only be used with certain types or modes, as shown in the following table:

	Modes			
Types	**read**	**create**	**update**	**append**

binary	positionfile()
	readfile()
	rewindfile()
		writefile()
		flushfile()
			positionfile()
			readfile()
			rewindfile()
			writefile()
			flushfile()
				writefile()
				flushfile()

text	readfile()
	rewindfile()
		writefile()
		flushfile()
			--	writefile()
				flushfile()

stream	positionfile()
	readfile()
	rewindfile()
		writefile()
		flushfile()
			positionfile()
			readfile()
			rewindfile()
			writefile()
			flushfile()
				writefile()
				flushfile()

Permissions

These statements are available to all users. When you create a file with the openfile() statement, the file is owned by you (the session user name).

Any operating system access protections apply to the files. For example, if the session user does not have permission to create a file in the specified directory, the openfile() statement fails.

Inquiring About and Setting Applications Status

4GL provides several statements and a function to retrieve information about an application that is running:

inquire_sql (statement)

This statement provides information about the result of query statements. It also provides information about the database session.

inquire_forms (statement)

This statement provides information about the status of various aspects of the Forms Runtime System (FRS).

dbmsinfo (function)

This function is used in a select statement to retrieve system information.

Also, you can use the set_sql, set_forms, and set_4gl statements to control various features of the application that is running.

Using the Inquire_sql Statement

The inquire_sql statement provides several types of information about the state of the application at run time:

- The number of rows affected by the last query statement
- The error number for any error that occurred on the last query
- The associated error text if an error occurred on the last query
- The session identifier for the current database connection.
- The connection name specified for the current database connection.

The following section discusses the rowcount, errorno, and errortext values returned by inquire_sql.

You also can use the inquire_sql statement to retrieve information about database events that have been raised. For a detailed discussion of using the inquire_sql statement with database events, see 4GL Statement Glossary (see page 935).

Use rowcount to determine how many rows were found by a select statement, added by an insert statement, changed by an update, or deleted by a delete statement. For example:

```
partstbl := select * from parts;
inquire_sql (rcount = rowcount);
if rcount < 1 then
  message 'No records were found';
  sleep 3;
endif;
```

When used in conjunction with select statements, the following rules apply:

- A singleton select always causes inquire_sql to return a value of 0 or 1 for rowcount.
- For attached queries, the value is the number of rows actually viewed by the user, as controlled by the next statement in the submenu.

Use errorno and errortext to check the results of a query operation, as in this example:

```
delete from employee
  where empnum = :empnum;
inquire_sql (errno = errorno,
  txt = errortext);
if errno != 0 then
  message 'Delete error ' + char(:errno) + ' ' + :txt;
else
  commit;
endif;
```

Use inquire_sql after the query statement but before commit. For more information about errors returned by inquire_sql and a complete list of its parameters, see the section Inquire_sql.

Using the Set_sql Statement

The set_sql statement lets you control various aspects of database behavior; for example, how errors are reported.

See Set_sql (see page 1154) for a detailed description of the set_sql statement.

Using the Inquire_forms Statement

The inquire_forms statement lets you retrieve information at run time from the FRS, which manages many aspects of the forms displayed as an application runs.

For a large set of form objects and for the FRS itself, Ingres provides a set of constants containing information you can use to perform conditional processing, error handling, determine whether a field or column is derived, and so on. For each type of object, the inquire_forms statement returns a different set of constants. See Inquire_forms (see page 1031) for more details. Inquire_forms does not operate on arrays.

The examples below show how to use inquire_forms to determine cursor location during an application. The first example finds the field (in the current form) on which the cursor currently rests:

```
inquire_forms field ''
  (fname = name);
```

In the following example, inquire_forms determines the row and column position of the cursor so that a message can be displayed beneath the current cursor location:

```
inquire_forms frs (row=cursorrow,
  col=cursorcolumn);
row = row + 1;
message 'No employee found with that name'
  with style=popup (startcolumn= :col,   startrow=
  :row);
```

Using the Set_forms Statement

The set_forms statement allows you to control many of the features of the FRS while the application is running with the statement:

```
set_forms
```

Most set_forms statement variants have a corresponding inquire_forms variant. This allows you to both set and query on the FRS features while the application is running. For example, use set_forms to turn on the blinking feature in a field to remind a user to enter a value, as in this example:

```
set_forms field dataform
  (blink (name) = 1);
```

Use another set_forms statement to turn blinking off, after the user enters the value:

```
set_forms field dataform
  (blink (name) = 0);
```

Tracking Changes to Form Fields and Rows

Changes made for each form, field or row are tracked with a *change bit* or *change variable*. Change bits are maintained for each simple field on a form and for each intersection of row and column in a table field and data set. The change bits are:

- Cleared (set to 0); used in the following cases:
- At the start of a display loop
- When the application places a new value into the field
- When the form, field, or row is set or cleared by the program
- When another frame or procedure to which the field or column has been passed by reference returns
- Set (set to 1), used whenever data is entered or re-entered by the user

An application can perform its own validations on values in fields, forms, and rows by checking the change bit. This reduces the number of times the value itself needs to be validated.

To set or clear the change bit, use the set_forms form, set_forms field, or set_forms row statements. The following example clears the change bit for salary on the *addemp* form:

```
set_forms field addemp (change(salary) = 0);
```

The following example clears the change bit for the "start_date" column in the second row of the "emp_hist" table field on the "addemp" form:

```
set_forms row addemp emp_hist 2
  (change (start_date) = 0)
```

To inquire about the change bit, use the inquire_forms form, inquire_forms field, or inquire_forms row (see page 1035) statements. For example:

```
inquire_forms field addemp
  (integer_variable = change(salary));
```

This statement inquires about changes to the salary field on the Addemp form. The *integer_variable* is set to 1 if salary is changed. Change is a keyword in the statement.

The following example uses the change bit when a table-field column is involved. It checks the "start_date" column for the current row in the "emp_hist" table field. This approach can also be used in an unloadtable loop.

```
inquire_forms row addemp emp_hist
  (integer_variable = change(start_date));
```

To track changes made to the form as a whole, use the inquire_forms form statement with the change FRS constant. For example:

```
'Save' =
begin    /* Save changes to fields */
  set_forms form (change = 0);
end
'End' =
begin    /* Call procedure 'oktoend' */
  if oktoend() = 1 then
    return;
  endif;
end
```

```
/* Code for procedure 'oktoend' follows: */

procedure oktoend(anychange = integer,
  ans = varchar(1)) =
  begin
  inquire_forms form(anychange = change);
  if anychange = 1 then
    ans := prompt 'Changes were not saved.' +
      'Would you like to end anyway?'
      with style = popup;
    if lowercase(left(ans,1)) = 'y' then
      return 1;
    else
      return 0;  /* continue working */
    endif;
  else
    return 1;
  endif;
  end
```

After performing a save, the example uses set_forms to clear the change status to 0. This status is set to 1 whenever the user types over the data displayed on the form. Before exiting a form, an application can check for unsaved changes.

Using the Dbmsinfo Function

You can use the 4GL dbmsinfo function in a select statement to furnish information about the current Ingres process, such as the Ingres release, the transaction state, or the username. Use this syntax:

```
select fieldname = dbmsinfo ('request name');
```

where *fieldname* is the name of a field into which the dbmsinfo result is retrieved, and *request name* is a quoted string corresponding to a Ingres constant. The following assignment statement displays the release number of the current Ingres release in the "charfld" field on the form.

```
select charfld = dbmsinfo('_version');
```

This example displays the current user's Ingres user name:

```
select charfld = dbmsinfo('username')
```

To compare the values of two dbmsinfo results, you must select the results into fields or variables. The next example assigns the request names *dba* and *username* to the local variables *user1* and *user2* and compares them in an if-then statement to determine if the current user is the DBA of the database. If so, the statements after the then clause are processed.

```
select user1 = dbmsinfo('dba'),
  user2 = dbmsinfo('username');
if user1 = user2 then
/* perform actions */
else
  message
  'You are not the DBA of this database';
  sleep 3;
endif;
```

The dbmsinfo function is not accessible directly through 4GL. If you use it outside of a database access statement, Ingres generally returns the empty string. No error message is issued.

For more information on the dbmsinfo function, see your query language reference guide.

Multiple Session Connections

ABF and Vision applications can connect to zero, one, or more database sessions. An application can open an initial session and, with subsequent connect statements, open additional sessions with the same or different databases.

By default, an ABF or Vision application connects on start up to the database from which the application was linked.

You can override the default when you run the application image. Run the image with the -d or -database flag to start the application with a connection to the specified database.

You can also start an application without a database connection, by running the image with the -nodatabase flag. Use the -nodatabase flag if the application does not require access to a database, that is, if the application has no database statements, QBF frames, report frames, or uncompiled forms.

If you start the application without a database, you can later connect to a database with 4GL code. The connect statement must precede any statements that access the database.

Using multiple sessions in ABF and Vision applications is similar to using multiple sessions in embedded SQL.

Connecting to Multiple Sessions

To open a session other than the initial session, you must issue the connect statement. To identify individual sessions in a multiple-session application, you assign a connection name and/or numeric session identifier when you issue the connect statement.

You can create multiple sessions that connect to the same database; for each connection, you can specify different runtime options, including user name. For details, see the Connect section.

The *current* session is established when an application issues a connect statement or switches sessions (using the set connection or set_sql(session) statements). If an error occurs when an application attempts to open a session, the application is connected to the previous active session.

Identifying Sessions

The connect statement enables you to assign each session a numeric session identifier and a connection name. The numeric identifier must be a positive integer; the connection name must be no longer than 128 characters. For details, see the section Connect.

Switching Sessions

To switch sessions using a numeric session identifier, use the set_sql(session) statement. To switch sessions using the connection name, use the set connection statement. To determine the numeric session identifier for the current session, use the inquire_sql(session) statement. To determine the connection name for the current statement, use the inquire_sql(connection_name) statement.

Note: After an application switches sessions, the error information obtained from the inquire_sql statement is not updated until an SQL statement has completed in the new session.

Disconnecting a Session

To disconnect from the current session, the application issues the disconnect statement. To disconnect a session other than the current session, specify the numeric session identifier or connection name. The disconnect all statement disconnects all sessions started with the connect statement. It does not disconnect any Ingres-generated sessions, including the initial or default session. Ingres-generated sessions are disconnected when the application exits.

After an application disconnects from the current session in a multi-session application, the application must issue the set connection, set_sql(session), or connect statement to establish another current session before issuing any database statements. If no current session is in effect when an application issues a query, Ingres returns an error.

Multiple Sessions and the DBMS

The DBMS treats each session in a multiple-session application as an individual application. When you are creating multiple-session applications, keep the following points in mind:

- Be sure that the server parameter connect_limit is large enough to accommodate the number of sessions required by the application. For details, see the *System Administrator Guide*.
- In a multiple-session application, the application can encounter deadlock against itself. For example, one session can attempt to update a table that was locked by another session.
- An application can also lock itself out in an undetectable manner. For example, if a table is updated in a transaction in one session and then selected from in another transaction in a second session, the second session waits indefinitely.

Built-In Frames and Procedures

You can perform a simple function or return a value in expressions by calling built-in frames and procedures with 4GL code. Built-in frames and procedures provide new features for your application without requiring new 4GL syntax or statements. You can override them with your own frames and procedures with the same names.

The following list briefly describes the 4GL built-in frames and procedures. For more information, see the name of each frame or procedure in 4GL Statement Glossary (see page 935).

ArrayAllRows()

Returns the total number of records in an array, including those marked deleted

ArrayClear()

Removes all records from an array, including any set to be deleted

ArrayInsertRow()

Inserts a record into an array at the record number you specify. The procedure renumbers the record that previously was at that position and the records that follow it.

ArrayFirstRow()

Returns the number of the first deleted row in an array

ArrayLastRow()

Returns the number of undeleted records in an array

ArrayRemoveRow()

Permanently removes a specified record from an array

ArraySetRowDeleted()

Marks the specified record of an array as deleted

Beep()

Beeps or rings the terminal bell if the terminal has one (some terminals flash the screen instead of beeping). It accepts no parameters.

CommandLineParameters()

Lets you retrieve into an application values that the user has entered on the command line when starting the application. The look_up() frame displays a pop-up frame with values from which the user can select to enter into a form field. These values can be based on a database table or an array.

Find_record()

Prompts the user for a value and searches for it in a table field column. Find_record() accepts keyword parameters to specify the form, table field, column, and an optional search value, or prompts for the search value.

Help_field()

Provides help on a field through the Field menu item of the help_forms statement submenu. The procedure does not accept parameters.

Sequence_value()

Returns a new sequence value (or range) for a database column. The sequence_value() function is a positive integer that is automatically incremented whenever a value is fetched. You can use the sequence value as a surrogate key assigned to a row of a relation, or as a join attribute between relations.

How Database Events Are Handled

4GL database event statements allow an application to notify other applications that a specific event has occurred. The other application then can use that information to perform some action that you designate.

The sequence for handling a database event is:

1. A *sending application* or database procedure raises the event.
2. The *Ingres DBMS Server* notifies other applications, called receiving applications, that the event has occurred.
3. A *receiving application* takes some action based upon the event. (The receiving application can be the same as the sending application.) The receiving application must:
   - Be registered to receive notification when the event is raised

     The server performs this notification by placing the event into an *event queue* within the receiving application.
   - Extract the event from the queue
   - Retrieve any desired information about the event

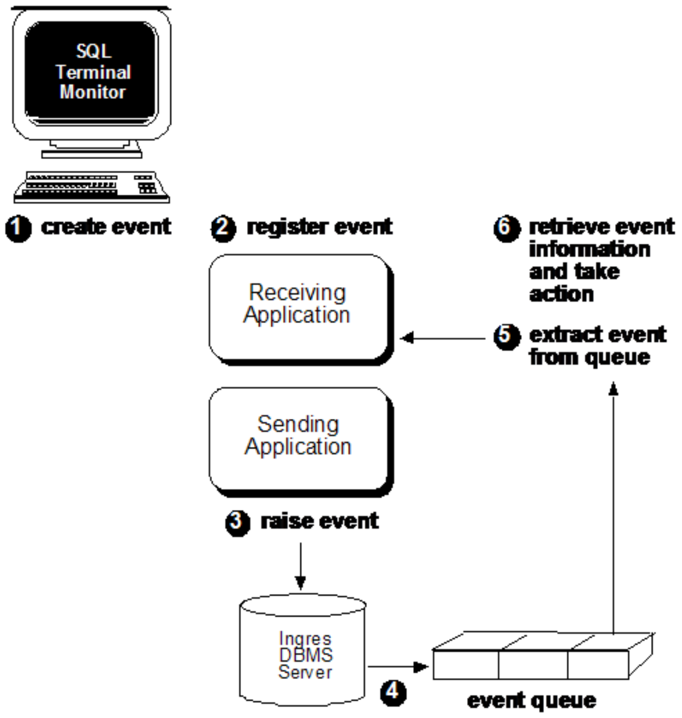
You can designate any program code within an application as a database event. For example, within an inventory control application, you can raise an event when an update causes the supply of a part to fall below a specified amount. A second application then is notified to reorder the part.

Event Handling Sequence Example

The following figure presents an overview of the database event handling process. The steps that follow use the example of two applications:

- The sending application is the Manufacturing application that, among its other functions, keeps track of the raw parts used in the manufacturing process.
- The receiving application is the Inventory control application that keeps track of supplies and orders them as necessary.

The following diagram illustrates the database event handling sequence:



The steps of the sequence are as follows:

1. You use the SQL create dbevent statement in the Terminal Monitor to create an event. For example, to create an event to be raised when supply of a part is low:

   ```
   create dbevent part_low
   ```

2. You register the event in your receiving application. For example, to register the "part_low" event in the Inventory application:

   ```
   register dbevent part_low
   ```

3. You place a raise dbevent statement after the appropriate code in your sending application. For example, to raise the "part_low" event in the Manufacturing application after an update that causes the quantity of a part to fall below 100:

   ```
   update parts
   set quantity = :quantity
   where part_no = :part_no;
   if quantity < 100 then
   raise dbevent part_low;
   endif;
   ```

4. When the supply of a part falls below 100, the raise dbevent statement is executed. The DBMS Server is notified that the "part_low" event has been raised. The server places the event into the event queue for the Inventory application.
5. An on dbevent activation block within the Inventory (receiving) application checks the event queue and extracts any events (for which it is registered) that have been raised.
6. An inquire_sql statement within the event activation block retrieves information about the event that has been raised. If it is the "part_low" event, the Inventory application takes the following actions:
   - Calls the "reorder" procedure that generates a purchase order for the part
   - Sends a mail message to the shop supervisor

The following is a sample of the 4GL code to accomplish steps 5 and 6:

```
on dbevent =
begin
  inquire_sql (ename = dbeventname,
               etime = dbeventtime);
  if ename = 'part_low' then
    callproc reorder;
    call system 'mail/subject = "Part ' +
    varchar(part_no) + ' was reordered at ' +
    varchar(etime) + '." FOREMAN';
  endif;
end
```

Event Handling Statements

The following table provides brief descriptions of the 4GL statements to enable the event handling sequence, and describes where to place the code for each statement.

See the *SQL Reference Guide* for detailed syntax and descriptions of the major statements involved in handling database events.

Statement	**Purpose**	**Location**
create dbevent *event_name*	Creates the named event in the DBMS Server.	In a separate "set-up" application, or in the Terminal Monitor with the SQL version of the statement. (The create dbevent statement must appear outside of the application that raises the event, because you create an event only once.)
drop dbevent *event_name*	Lets the event owner destroy the event.	Same location as above.
raise dbevent *event_name* [*event_text*]	Notifies the DBMS Server to send a message that the named event has been raised; this message is placed in an event queue for each application that is registered to receive events.	Within the sending application, immediately after the code that describes the event (and before any transaction-handling statements, such as a commit).
register dbevent *event_name*	Notifies the DBMS server that an application is eligible to retrieve the named event.	Within the receiving application, before the code (get dbevent or on dbevent activation block) that receives the event. The register dbevent statement is valid for the duration of the session, or until you issue a remove dbevent statement for the event.
remove dbevent *event_name*	Removes an application's registration for the named event.	Within the receiving application, when you no longer want the application to be able to receive the event.

Statement	Purpose	Location
get dbevent [with nowait | wait = *wait_interval*]	Checks an application's event queue; if it finds any events, it retrieves from the queue the next event for which the application is registered.	Within the receiving application. You must issue a get dbevent statement (or use an on dbevent activation, described below) before you can retrieve specific information about an event.
on dbevent *activation_block*	Performs an implicit get dbevent statement when a user activation is executed; lets you specify statements to be executed if an event is found.	Within any frame or display submenu that might be in control when an event is raised. (You cannot have an event activation without some other activation in the same frame or display submenu.)
inquire_sql event_info	Retrieves specific information about an event that has been received with a get dbevent statement or on dbevent activation.	Within the receiving application, following a get dbevent statement or within an on dbevent activation block. You must issue an inquire_sql statement before the next get dbevent or on dbevent activation, because the information about the last event is overwritten by the next event the application receives.

Putting It All Together: 4GL Specification

The following example shows how 4GL statements can be combined in a user-frame specification. The 4GL specification sets up the operations menu and defines all the operations and activations in the frame.

Coding an Operation

The specification contains sets of statements which cause operations or activations to take place. The following example shows a menu activation:

```
'Exit' =
begin
  message 'Exiting...';
  sleep 3;
  exit;
end
```

In this case, the word "Exit" appears as part of the menu for the frame. The user executes the statement sequence by choosing Exit.

"Exit" is a reserved word, so it is given quotation marks in the example. Quotes around the menu item name are not required unless it contains a reserved word. However, it is a good idea to use quotes to avoid any possible conflict.

Indicate the beginning and end of the statement series with the keywords begin and end, as shown in the example. You can use braces in place of the begin and end statements:

```
'Exit' =   /* Exit the application */
begin
  message 'Exiting . . .';
  sleep 3;
  exit;
end
```

Following the begin and preceding the end are the statements that accompany the menu operation. These statements are separated from each other by semicolons (;), which are required. You can write the statement list on several lines. Do not follow begin and end with semicolons.

Comments

A comment line follows the first line of code of the Exit menu operation in the example above. A comment is an arbitrary sequence of characters bounded by "/*" on the left and by "*/" on the right:

```
/* This is a comment */
```

Comments are a useful way to label the parts of the specification clearly. You can place these wherever you can place a blank space or carriage return. A comment can extend over more than one line, but cannot be nested.

Using Activation Statements

Use activation statements to set up the way the application user manipulates the frame. These statements specify a set of 4GL statements that executes each time the user chooses a menu operation, presses a function key, or attempts to move into or out of a specific field, or when a time-out occurs. At the end of each activation, you can use the resume statement to return the cursor to any specified position on the frame.

Types of 4GL Activations

An activation in 4GL is an initialize statement or a field, key, menu, time-out, or database event activation. An operation in 4GL is defined by one or more of the following activations:

- An initialization occurs when a frame starts up, before the form is displayed. Use initializations to set or alter the contents of the form before the user interacts with it.
- A menu activation occurs when the user chooses a menu operation.
- A key activation occurs when the user presses an FRS key.
- A field activation occurs when the user enters or leaves a field.
- A timeout activation occurs when a preset time-out period has elapsed without keyboard activity.
- A database event activation occurs when one of the other activations causes the presence of a database event to be detected.

The initialization section is optional, but if it does appear, it must be first. Other operations can appear in any order. The following sections describe the overall structure of a 4GL source file.

Using an Initialization Section

The initialization section of a specification includes the initialize and declare statements. Use the initialization section to:

- Include a set of 4GL statements to execute when the frame starts up, before the form is displayed. These can include setting the mode of the form, defining key operations, and assigning initial values to fields and variables.
- Carry out an initial processing sequence before the application user does anything.
- Declare local variables and hidden columns which you can use throughout the specification file for the frame.

The syntax below shows these statements.

```
initialize [( {localvariable = typedeclaration,} )] =
[ declare { localvariable = typedeclaration ,} ]
  begin
    /* statements */
  end
```

Variables declared in the initialize portion can have values passed into them by a calling frame while those declared in the declare section cannot.

The following example sets the displayed field *selection* to a value of 1, sets the displayed field *lname* to the string "Last," and creates the local variable *total* and sets it to 0:

```
initialize =
  declare total = smallint not null
begin
    lname := 'Last';
    selection := 1;
    total := 0;
end
```

The begin and end keywords in the statement enclose a group of statements that are to be executed every time the frame is called. For syntax and further considerations, see the section Initialize.

Menu Activations

Use a *menu activation* to:

- Define the name of the menu operation for display on the user's screen
- Specify the series of 4GL statements that executes each time the operation is selected
- Associate menu and key activations

The syntax for defining a menu operation is as follows:

```
'menuitemname'
 [ ( [validate = value ] [, activate = value]
 [, explanation = string] ) ]
[, key frskeyN
 [ ( [validate = value ] [, activate = value] ) ] ] =
 begin | {
   statement; {statement;}
 end | }
```

The *menuitemname* is the name that appears on the menu line. Quotes surrounding the *menuitemname* are optional except where the name is a 4GL keyword. It is a good idea to use them routinely to avoid any possible name conflicts. Enclose all reserved words and single characters in quotes. You can specify *menuitemname* through a global constant, preceded by a colon.

The frskey*N* is the FRS key designation (*N* is a number from 1 to 40) for a key that can activate the menu operation. Depending on the key mapping in use, each menu name is displayed in the window, and can be followed by a label indicating the activating key.

Menu operations are assigned default key mappings based on their sequence in the frame source code. You can use key frskey*N* to override these default key mappings by mapping to an FRS key you select. In this case, the label for the control or function key mapped to that FRS key is displayed. For more information on key mapping, see *Character-based Querying and Reporting Tools User Guide.*

It is helpful to provide several ways to execute the same set of 4GL statements. In the following example, both the Help menu operation and the key associated with FRS key 1 display the help file for the start frame of the sample application:

Windows:

```
'help', key frskey1 =
 begin
  help_forms (subject =
    'Start Frame Help Information',
    file = '\usr\admin\files\personnel.hlp' c:)
end
```

UNIX:

```
'help' (activate = 1), key frskey1 (activate = 1) =
 begin
  help_forms (subject =
    'Start Frame Help Information',
    file = '/usr/admin/files/personnel.hlp')
end
```

VMS:

```
'help' (activate = 1), key frskey1 (activate = 1) =
begin
  help_forms (subject =
    'Start Frame Help Information',
    file = 'dra0:[usr.admin.files]personnel.hlp')
end
```

The *value* for validate or activate is 0 or Off to turn the option off, or 1 or On to turn the option on.

- Validate. If you turn validation on, the current field (where the cursor is when the operation is selected) is validated according to criteria established in the form definition window before the key or menu activation block runs. If the field fails the check, a message appears, and the cursor is positioned back on the field without executing the operation. No validation is performed if the form is in query or read mode.
- Activate. If activation is on, the activate block for the current field is executed before the key or menu activation block runs. No exit activation is performed if the form is in read mode.

If you specify both a menu item and an FRS key for your activation block, you must indicate the validate or activate keyword separately for each.

If validate or activate is not specified, the default action is controlled by whether it has been turned on globally for menu operations through a set_forms frs statement.

If an after field activation block exists, it must include a resume next statement or the menu activation block does not run.

Use the explanation clause to enter a string that describes a frame menu item. (You can specify this string through a global constant, preceded by a colon). The description appears in a table field accessed by the help/keys operation. When explanation is not used, the Explanation column of this table field is blank for the menu item. For more on this clause, see the section Helpfile Help_Forms.

Any sequence of 4GL statements can appear within a menu definition. You must place them between the keywords begin and end or within braces. The statements are executed whenever the user chooses the menu operation. For example:

```
'Increment' (validate = 1) =
begin
    age := age + 1;
    message 'Age has now been changed';
    sleep 3;
end
```

In this specification, the menu operation Increment appears in the user's window. Whenever the user chooses Increment, Ingres validates the current field according to the criteria established when the form was created with the ABF FormEdit operation. Validation takes place because of the optional validate = 1 clause.

- If the value in the field is valid, the value in the *age* field is incremented, and the message "Age has now been changed" appears in the window.
- If the field fails the validation check, a message is displayed to the user and returns the cursor to the current field without performing the Increment operation.

For further details on field validations, see *Character-based Querying and Reporting Tools User Guide.*

Field Activations on Entry and Exit

Use a *field* activation operation to specify a series of 4GL statements to be executed when the cursor moves into or out of a specific field on a form. The syntax is:

```
[ before | after ] field
  simplefieldname|all|tablefieldname.columnname
  |tablefieldname.all
{, [ before | after ] field
  simplefieldname|all|tablefieldname.columnname|
  tablefieldname.all} =
  begin | {
    statement; {statement;}
  end |}
```

Note that an activation can be shared by more than one field. Also, you cannot perform activations on an entire table field; you must indicate specific columns (or use the keyword all).

The keywords before and after determine whether the activation takes place when the cursor enters the field or as the cursor exits. These keywords are optional. If omitted, the activation occurs upon exit from the field (after).

The keyword all sets an activation on all the simple fields in the form or columns in the table field. If a frame has activations on all and on an individual field or column, the last activation is the one that is executed. This applies to both before and after field activations.

Any sequence of 4GL statements can appear within the definition of a field activation. The 4GL statements that accompany the field definition are executed whenever the user leaves or enters the field, depending on the type of activation:

- Field exit activation (the default) is useful for data validation or control flow modification.
- Field entry activation is useful for activities such as highlighting the current field or providing field help information.

If specified, exit (keyword after) activations are performed in fill, update, and query mode; they do not occur in read mode. Entry activations occur in all modes including read mode (for a form or table field).

You can enable or disable activations globally for the application using the set_forms frs statement. By default, entry and exit activations are enabled.

Note: Be careful when writing entry activations, as an endless chain of entry activations can result. For example, if fields A and B have entry activation code containing a resume to each other, an infinite loop results once the user enters either field.

When the following statement block is used with an exit activation (keyword after), Ingres checks to see if the value in the field is zero or Null when the user leaves the *empnum* field. If so, it issues the error message and returns the cursor to the empnum field. If not, it retrieves the employee's name from the database and positions the cursor at the *lname* field.

```
field empnum =
begin
  if empnum = 0 or empnum is null then
    message 'Please enter an employee number';
    sleep 3;
  else
    empform := select fname, lname
      from employee
      where num = empnum;
    resume field 'lname';
  endif;
end
```

In an exit activation, the last statement to be executed must be a resume statement. Without the resume statement, the cursor remains on the same field after all statements are executed, and does not exit to another field.

In general, entry activations follow the same rules as other activations and can be combined with them.

You cannot use field activations in operations lists for the run submenu statement, because no fields are accessible while the run submenu statement is executing.

Note: If the cursor is in a column for which an activation is defined, the activation causes the loss of multi-row scrollup and scrolldown capabilities.

Key Activations

Use a *key activation* operation to define the results of pressing a specific key on the keyboard. The operation is specified *not* for a particular function or control key, but for an *FRS key*, which is then mapped to the function or control key.

The syntax is:

```
key frskeyN [( [ validate = value ]
    [, activate = value] )] =
begin | {
    statement; {statement;}
end | }
```

The frskey is the FRS key designation (*N* is a number from 1 to 40) for a key you want to activate.

The *value* for validate or activate is 0 or off to turn the option off, or 1 or on to turn the option on.

Validate

If you turn validation on, the field where the cursor resides is validated according to criteria established in the ABF FormEdit operation before the key activation block runs.

If the field fails the check, a message appears, and the cursor is positioned back on the field without executing the operation. No validation is performed if the form is in query or read mode.

Activate

If activation is on, the activate block for the current field is executed before the key activation block runs. No exit activation is performed if the form is in read mode.

- If validate or activate is not specified, the default action is controlled by whether it has been turned on globally for key operations through a set_forms frs statement.
- The field activation block must issue a resume next statement or the key activation block does not run. If no field activation block exists, the system runs the key activation block.

Any sequence of 4GL statements can appear after the key definition. You must place them between the keywords begin and end or within a pair of braces.

The 4GL statements that accompany the key activation definition are executed whenever the user presses the control or function key mapped to the specified FRS key. For example, the following statements start the Help facility with the personnel.hlp file when the user presses the key mapped to the FRS key 1.

Windows:

```
key frskey1 =
begin
  help_forms (subject =
    'Start Frame Help Information',
    file = '\usr\admin\files\personnel.hlp' c:)
end
```

UNIX:

```
key frskey1 =
begin
  help_forms (subject =
    'Start Frame Help Information',
    file = '/usr/admin/files/personnel.hlp')
end
```

VMS:

```
key frskey1 =
begin
  help_forms (subject =
    'Start Frame Help Information',
    file = 'dra0:[usr.admin.files]personnel.hlp')
end
```

Timeout Activations

Use a *timeout activation* operation to specify the actions to be taken if timeout occurs. A timeout block begins with the keywords on timeout. The syntax is:

```
on timeout =
  begin | {
    statement; {statement;}
  end | }
```

The timeout feature enables you to control how long the application waits for requested input from a user. If the user fails to make a menu choice or respond to a prompt or message within the specified time, the timeout occurs, returning control to the application. The timeout sequence defined by the on timeout block is then activated.

You must repeat the on timeout block in all frames and submenus where activation is to occur on a timeout. If a timeout occurs without a timeout activation, the default is to complete the statement that is waiting for input. If no statement is waiting for input, such as a cursor positioned in a field, a Return is performed.

To enable timeout, include a set_forms statement (for example, in the initialization block) that specifies the length of the timeout period in seconds. For example:

```
set_forms frs (timeout = 60);
```

Once set, the timeout period applies to all display loops, prompts, messages, and all error messages requiring a Return for confirmation.

The timeout period remains in effect until another set_forms statement changes the period's length or sets it to zero, which effectively turns timeout off. You can place the statement inside or outside of a display loop, or inside a nested display loop. Once executed, it affects the application globally.

Timeouts set within an application do not affect any programs called by the application. To impose timeout limits on called programs, do so within the code of the called program itself.

The following timeout block clears the screen (of possibly confidential information) whenever user input has ceased for 10 seconds. The user must input the password "master" to reactivate the screen:

```
on timeout =
begin
  clear screen;
  set_forms frs (timeout = 0);
  pass_var := '';
  while pass_var <> 'master' do
    pass_var := prompt noecho 'Password: ';
  endwhile;
  set_forms frs (timeout = 10);
  redisplay;
end
```

Timeout periods set to a small number (less than 10 seconds) cannot be handled reliably, depending on the system and its current load. The following example allows a much longer timeout. It blanks the screen after 9 minutes of inactivity, then redisplays the screen whenever Return is pressed:

```
on timeout =
begin
  clear screen;
  set_forms frs (timeout = 0);
  pass_var := prompt '';
  set_forms frs (timeout = 540);
  redisplay;
end
```

Database Event Activations

Use a *database event activation* block to specify the actions to be taken when another activation signals that a database event has been raised. The code in the activation block is executed when the presence of the new event is detected.

To retrieve information about the event that caused the activation, you must place an inquire_sql statement within the activation block. Include any of the inquire_sql event-related parameters that are needed. (See the section Inquire_sql for more information on the inquire_sql statement.)

The syntax of an event activation block is:

```
on dbevent =
begin | {
[  inquire_sql ([var = dbeventname, ]
                [var = dbeventowner, ]
                [var = dbeventdatabase,]
                [var = dbeventtime,]
                [var = dbeventtext]);]
   statement; {statement;}
end | }
```

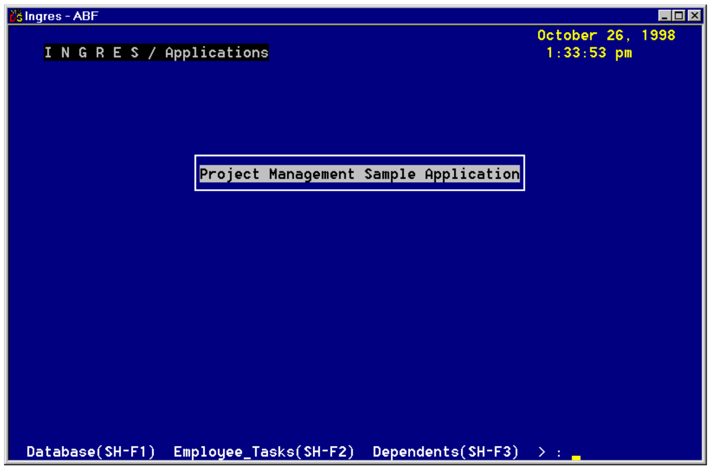
You must accompany a database event activation with a field, key, menu, or timeout activation in the same frame or display submenu, because the search for a database event in the event queue is made only when some other activation is about to be initiated or completed. If a database event is found, the statements associated with the database event activation are executed before any statements for the other activation (if it was about to be initiated).

You can use database event activations within a frame (outside any submenus) or within a display submenu statement, but not with a run submenu statement. You cannot use any variants of the resume statement other than resume without any operands.

See How Database Events Are Handled (see page 914) for a detailed description of event handling.

Example

This chapter concludes with a complete 4GL specification for a sample start or top frame. The following figure shows the opening frame of the Project Management application. This is the same user frame, with an expanded menu, that was introduced briefly in Overview of 4GL (see page 809).


Ingres - ABF
I N G R E S / Applications
October 26, 1998
1:33:53 pm
Project Management Sample Application
Database(SH-F1) Employee_Tasks(SH-F2) Dependents(SH-F3) > :

The menu for this frame was created using the following 4GL specification. The form was created using the ABF FormEdit operation. The 4GL specification pairs each menu operation with statements to be executed whenever the user chooses the operation. It also defines the key activations for this frame.

```
/*
**
** top.osq; main frame for the projmgt
** application
**
**/

initialize =
begin
  message 'Starting the ABF Demonstration' +
  ' Application. . .';
  today = date('today');
  callproc startup;
  callproc timer_on(secs=2);
end
```

```
on timeout =
begin
  current_time=date('now');
end

Database =
begin
  callframe database;
end

Employee_Tasks =
begin
  set_forms field ''   (invisible(current_time)=1);
  redisplay;
  callframe emptasks;
  set_forms field ''   (invisible(current_time)=0);
end


Dependents =
begin
  callproc timer_off;
  callframe empdep;
  callproc timer_on(secs=2);
end

Experience =
begin
  callproc timer_off;
  callframe experience;
  callproc timer_on(secs=2);
end

Mail =
begin
  call system 'mail';
end
```

```
'4GL', key frskey16 =
begin
  callproc timer_off;
  helpfile 'Top Frame 4GL'
    '$dra0:[usr.admin.files.abfdemo]top.osq';
  callproc timer_on(secs=2);
end

'Help', key frskey1 =
begin
  callproc timer_off;
  help_forms (subject =
    'Top Frame Help Information',
    file = '$dra0:[usr.admin.files.abfdemo]top.hlp');
  callproc timer_on(secs=2);
end

'Quit', key frskey2, key frskey3 =
begin
  exit;
end
```

In the example, activations specify the main functions of the frame, as follows:

- The initialization section calls a separate procedure (using the callproc statement) to complete the initialize sequence.
- Menu activations set up the menu selections (Database, Employee_Tasks, Dependents, Experience, Mail, 4GL, Help, and Quit).
- Key activations provide an alternate way of selecting some of the menu operations.

The callframe statements suspend the operation of the current frame to call another form and operations menu or start up a specific operation such as looking up data or running a report.

The call system statement is executed when you choose the Mail operation. It starts the operating system's mail utility and then returns to this frame when you exit from Mail.

The helpfile statement is executed when you choose the 4GL operation or presses the key associated with FRS key 16. It displays the 4GL source code file for this frame.

The help_forms statement is executed when you choose the Help operation or presses the key associated with FRS key 1. It displays the Help file for this frame.

The exit statement is executed when you choose the Quit operation or press the key associated with FRS key 2 or 3. It ends the application and returns the user to the operating system.

An on timeout block is used to display the current time at the end of the 2-second timeout period.

This frame is part of the Project Management Sample Application, presented in the ABF part of this guide.

Chapter 21: 4GL Statement Glossary

This section contains the following topics:

This chapter describes each 4GL statement, providing the purpose, syntax, and examples of the statement. You can substitute components of some 4GL statements at run time, either through expressions or through 4GL names. For rules governing runtime substitution, see Using 4GL (see page 813).

The full range of statements available to you in a 4GL program includes interactive SQL statements as well as those that can be used with a distributed database.

In most cases, the query statements in 4GL correspond very closely to their interactive versions. Using SQL with 4GL (see page 937) describes the differences.

Reserved words for SQL and QUEL are listed in Keywords (see page 1317). See the appropriate query language reference guide for further information on additional reserved words or SQL statements.

Text conventions used in presenting the syntax of statements in this chapter are described in Introduction (see page 31).

4GL names are underlined in syntax statements.

Using SQL with 4GL

In a 4GL program, you can use interactive SQL statements as well as those that can be used with a distributed database. All statements have the same syntax as the corresponding interactive SQL statement, with the exceptions noted below. You can use 4GL names to specify many components in SQL statements. For rules on using 4GL names, see 4GL Names (see page 847).

The following statements are not supported in 4GL:

- create procedure
- create schema
- drop procedure
- drop schema
- get data
- put data

The following statements have syntax or usage in 4GL that is different from interactive SQL. See the statement descriptions in this chapter for detailed information:

- connect
- delete
- disconnect
- execute immediate
- execute procedure
- insert
- select
- update

4GL supports Ingres SQL data types, with a few exceptions. For more information about 4GL data types, see the section, Data Types for Simple Local Variables and Hidden Columns (see page 817).

ArrayAllRows()

Counts records in an array.

Syntax

```
returnfield = [callproc] arrayallrows(arr)
```

returnfield

Specifies the name of field to which the result is returned. Integer.

arr

Specifies the name of an array

Description

The arrayallrows() built-in function returns the total number of records in an array, including those marked deleted.

The procedure returns 0 if the array is empty and contains no deleted rows.

Example

To count the total number of employee records in the array *emparray*, including deleted records:

```
total = callproc
  arrayallrows (emparray);
```

ArrayClear()

Clears the records from an array.

Syntax

```
[returnfield =] [callproc] arrayclear(arr)
```

returnfield

Specifies the name of field to which the result is returned. Integer.

You must include a value for returnfield if you omit the callproc keyword; otherwise it is optional.

arr

Specifies tghe name of an array

Description

The arrayclear() built-in procedure removes all array records, including any marked as deleted. The rows are completely removed, as with the arrayremoverow() function (described later in this chapter).

The procedure returns 0 on success or a non-zero number on failure. Failure occurs only in rare circumstances, such as trying to pass the procedure a parameter that is not an array.

Example

To clear all records from the array *emparray*:

```
callproc arrayclear (emparray);
```

ArrayFirstRow()

Returns the number of the first deleted record in an array.

Syntax

```
returnfield = [callproc] arrayfirstrow(arr)
```

returnfield

Specifies the name of field to which the result is returned. Integer.

arr

Specifies the name of an array

Description

The arrayfirstrow() built-in function returns the number of the first deleted record in an array.

When records are deleted, 4GL moves the deleted records to the beginning of the array and assigns 0 or negative sequence numbers to them (0, -1, etc.). This statement returns the number of the "first" deleted record (the one with the largest negative number). If there are no deleted records or if there are no records in the array (deleted or not), arrayfirstrow() returns a value of 1.

Example

To find the number of the first deleted row in the array *emparray*:

```
x = callproc arrayfirstrow (emparray);
```

ArrayInsertRow()

Inserts a record into an array.

Syntax

```
[ returnfield = ] [callproc] arrayinsertrow
 (arr, rownumber = integerexpr);
```

returnfield

Specifies the name of field to which the result is returned. Integer.

You must include a value for returnfield if you omit the callproc keyword; otherwise it is optional.

arr

Specifies the name of an array

integerexpr

Specifies an expression that indicates the record number of the new record

Description

The arrayinsertrow() built-in function inserts a record at the record number you specify and renumbers the record that previously was at that position and the records that follow it. This contrasts with the behavior of insertrow, which inserts a row after the position you specify.

The procedure returns 0 on success, or a non-zero number on failure. Failure occurs when the specified record is not found in the set of records in the array or immediately following the end of the array.

Example

To insert an employee record into the second position in the array *emparray*:

```
callproc arrayinsertrow
  (emparray, rownumber = 2);
```

This inserts a blank record at record 2. The previous record 2 moves down and becomes record 3, and so on. To insert data into the new record:

```
emparray[2].col1 = 1135;
emparray[2].col2 = "Anderson";
```

ArrayLastRow()

Counts non-deleted records in an array.

Syntax

```
returnfield = [callproc] arraylastrow(arr)
```

returnfield

Specifies the name of field to which the result is returned. Integer.

arr

Specifies the name of an array

Description

The arraylastrow() built-in procedure returns the number of undeleted records in the array you specify. It does this by counting the non-deleted records and returning the number of the record with the highest sequence number.

The procedure returns 0 if the array contains no non-deleted rows.

Example

To count the total number of employee records in the array *emparray*, and exclude any deleted records:

```
total = callproc arraylastrow (emparray);
```

ArrayRemoveRow()

Removes a record from an array.

Syntax

```
[ returnfield = ] [callproc] arrayremoverow
    (arr, rownumber = integerexpr)
```

returnfield

Specifies the name of field to which the result is returned. Integer.

You must include a value for returnfield if you omit the callproc keyword; otherwise it is optional.

arr

Specifies the name of an array

integerexpr

Specifies an expression that indicates the record number of the record to be removed

Description

The arrayremoverow() built-in procedure permanently removes the specified record from an array. The record no longer exists in the array; as far as calls to arrayfirstrow() and arrayallrows() are concerned, the record never existed. You can use this to remove records marked deleted. The records following the removed one are renumbered.

The arrayremoverow() procedure returns 0 on success, or a non-zero number on failure (that is, the record you specified does not exist).

Example

To permanently remove the third record from the array *emparray*:

```
callproc arrayremoverow
  (emparray, rownumber = 3);
```

To permanently remove the first two records from the array *emparray*:

```
callproc arrayremoverow
  (emparray, rownumber = 1);
callproc arrayremoverow
  (emparray, rownumber = 1);
```

Rownumber is equal to 1 in both statements because, after removing the first record, the old second record becomes the first.

ArraySetRowDeleted()

Deletes an array record.

Syntax

```
[ returnfield = ] [callproc] arraysetrowdeleted
    (arr, rownumber = integerexpr)
```

returnfield

Specifies the name of field to which the result is returned. Integer.

You must include a value for returnfield if you omit the callproc keyword; otherwise it is optional.

arr

Specifies the name of an array

integerexpr

Specifies an expression that indicates the record number of the record to be deleted

Description

The arraysetrowdeleted() built-in function marks the specified record deleted (this does not actually remove the record from the array). The _state attribute of the record is changed to "Deleted."

The record is moved to the beginning of the array, and its _record attribute is changed accordingly. The first record to be deleted becomes record number "0," the second record to be deleted becomes record number "-1," and so on.

The procedure returns 0 on success, or a non-zero number on failure (that is, the record you specified does not exist).

Example

To mark a record from the array *emparray* deleted (the record remains in the array):

```
callproc arraysetrowdeleted
  (emparray, rownumber = 2);
```

Assignment

Assigns values to simple and complex form fields.

Syntax

For direct assignments:

```
simpleobjectname  := | = expression
```

For query assignments:

```
complexobjectname  := | = querystatement [submenu]
```

The following are the parameters for the assignment statement. None of these are 4GL names.

simpleobjectname

Specifies the name of a simple field, a local or global simple variable, a table-field cell, or a simple record attribute. The data types of the expression and the object must be compatible. Cannot be the name of a derived field.

Write table-field cells as: *tablefieldname***.***columnname* (which refers to the row under which the cursor is positioned)
or: tablefieldname**[***integerexpression***].***columnname*

Write record attributes as: recordname.attributename
or: arrayname**[***integerexpression***].***attributename*

expression

Specifies the value the statement is assigning to *simpleobjectname*. Any legal 4GL expression. The data types of the *expression* and the *objectname* must be compatible. *Expression* can be a constant or the contents of a field, expression, table field, entire form, global or local record type, array or variable.

complexobjectname

Specifies the name of a form, table field, a table-field row, a record, or an array. Where the table field name and the form name are identical, write the table field as *formname***.***tablename*. Cannot be the name of a derived field.

querystatement

Specifies any legal select statement. See Select to a Complex Object (see page 1111).

submenu

Specifies a menu displayed for the duration of a select offering you options relating to each row of values

Description

The assignment statement takes a value produced by an expression and assigns it to a form, a table field in a form, a row or cell in a table field, a simple variable, a record, a record attribute, or an array. (You cannot assign values to a derived field or column.) The assignment statement has two variants, according to the type of field to which a value is assigned:

- In *direct assignments*, you can assign values to fields or simple variables one at a time. See Direct Assignments (see page 945).
- *In query assignments,* you can assign values to an entire form, table field, table field row, record, or array at once. See Query Assignments (see page 946).

"**:=**" or "**=**" are the two forms of the assignment operator. You can use either one. This operator assigns the value of the expression on the right of the statement to the object named on the left. The two must be of compatible data types.

A semicolon (;) statement separator is necessary if the statement is part of a sequence of statements.

For query statements, the optional *submenu* can be included in assignments to forms, table fields, records, and arrays. For more information about queries, see Select (see page 1094).

Direct Assignments

A direct assignment statement assigns a value into a simple field in a form, a simple local variable, a column in a table field, or the attribute of a record or an array record attribute. The value and the object must have Ingres data types that are compatible.

During character field assignment, 4GL truncates values that are longer than the field itself. Other data types must fall within their appropriate numeric range.

4GL does not automatically perform a validation check when an expression is assigned to a field. To check that the value conforms to the current VIFRED specification for the form, you can either:

- Issue a validate field statement
- Access the field and check for an error using inquire_forms frs

In the event of a validation check failure, return control to the user to correct the field input.

Query Assignments

This type of assignment statement uses a query to assign a list of values from the database to several or all simple fields on a form, to the rows of a table field, to record attributes, or to an array. In a query assignment, you use a select statement. You can optionally include a submenu.

For details on constructing queries, see Select (see page 1094).

Examples

The following examples show direct assignments.

Assign given values to the Name and Empnum simple fields of the form:

```
name := 'John';
empnum := 35;
```

Assign specified values to the columns Name and Age in the second row of the table field named Child. The integer expression "2" refers to a row in the table-field display in the window, not to a record in the underlying data set.

```
child[2].name := 'Sally';
child[2].age := 8;
```

Place the specified values in the Name and Age columns in the current row of the table field named Child:

```
child.name := 'Steven';
child.age := 11;
```

Assign specified values to the *name* and *salary* attributes of the record *employee*:

```
employee.name := 'John' ;
employee.salary := 50000 ;
```

Assign specified values to the name and address attributes in the third row of the array named parent. The integer expression 3 refers to a record in the array, not to a visible field on a form.

```
parent[3].name := 'Janet' ;
parent[3].address := 'New York' ;
```

The next examples show query assignments:

Assigns Name and Cost from the Objects table to the array *pricearr*, based on the color of the objects:

```
pricearr = select name, cost
  from objects
  where objects.color = 'RED';
```

The following statement retrieves values from the Projects table to simple fields on the form Empform. The value from the column Projname is assigned to the Task field, while the value from the Hours column is assigned to the Time field. As a simple field assignment, this statement reads one record from the database table.

```
empform := select task = projname,
  time = hours
  from projects;
```

The following **select** query reads multiple records selected from the database table Parts into a data set associated with the table field *partstbl*.

```
partstbl := select number, name, price
  from parts;
```

The examples below show query assignments to an array record and to a table field row. Note the differences in syntax.

The following example reads the columns *name*, *job_title*, and *ssno* from the Employees table for the record whose value for *ssno* is '555-55-5555'. They are read into the attributes *name*, *job_title*, and *ss* for the first record in the array *emparr*. This array has another attribute, *salary*, which is set to 0.

```
emparr[1] := select name, job_title,
    ss = ssno
  from employees
  where ssno = '555-55-5555';
```

This example reads the columns *name*, *job_title*, and *ssno* from the Employees table for the record whose value for *ssno* is '555-55-5555'. They are read into the table field columns *name*, *job_title*, and *ss* for the current row of the table field. The cursor must be in the table field for this query to succeed. This row of *emptbl* has another attribute, *salary*, which is cleared.

```
emptbl := select name, job_title,
    ss = ssno
  from employees
  where ssno = '555-55-5555';
```

Beep()

Causes monitor to beep or ring.

Syntax

```
callproc beep()
```

Description

The 4GL beep() procedure beeps or rings the monitor bell, if the monitor has one. Some monitors flash the screen rather than ring or beep. The beep() procedure accepts no parameters and returns no values.

Call

Calls an Ingres tool (subsystem) or another application, with parameters.

Syntax

```
call subsystem ([parametername = value
    {, parametername = value}])
```

subsystem

Specifies an Ingres tool or application

parametername

Specifies a parameter to be passed to the subsystem. This is often a synonym for a command line flag or the name of an Ingres object.

value

Specifies the value to be assigned to the parameter: any single-valued expression that evaluates to the data type of the parameter and that fits the context of the parameter.

Place string values in quotes. For a parameter with no value, place an empty string to the right of the equal sign. Fields specified must be non-nullable.

Description

The 4GL call statement lets you call an Ingres tool from within a user-specified frame in an application. When the application user exits from the subsystem, 4GL returns to the current form, and control passes to the statement following the call.

The *subsystem* is a 4GL name that you can specify at run time.

Before issuing a call *subsystem* statement, you must commit any ongoing multi-query transactions. If you do not issue a commit, 4GL automatically commits the transaction without a warning message before it invokes the subsystem.

For more information about the functions and flags of each subsystem, see the appropriate sections of this guide.

You can pass parameters of any legal type to the subsystem. 4GL converts the values to one of the base types of character, integer, decimal, or float, according to the following table:

4GL Type	Base Type
integer	integer
float	float
money	float
decimal	decimal
date	string (length=25)
c	string
char	string
text	string
varchar	string

These are the variants of the call statement:

call abf

Calls ABF

call application

Calls an application

call ingmenu

Calls the Ingres Menu

call isql

Calls Interactive SQL

call qbf

Calls QBF

call rbf

Calls RBF

call report

Calls the Report-Writer to run a report

call sql

Calls the SQL Terminal Monitors

call sreport

Calls the Report-Writer to store a report specification in the database

call vifred

Calls the VIFRED for creating or editing forms

call vision

Calls Vision

You use different parameters and values in a call subsystem statement depending on the subsystem or application. Parameter names for each variant are listed below. Many of these parameters have the same effect as command line flags used when you call the subsystem from the operating system.

For most subsystem calls, you can use the flags parameter to pass the values of any flags, including those that do not have defined parameter names, with the exception of the -u flag. You can use the -u flag only when you invoke a database at the start of the underlying application.

Within the string containing the flags, separate each distinct flag by a blank space or tab. For a list of available flags for each subsystem, see your query language reference guide.

The flags parameter does not exist for calls to application. In addition, calls to an interactive query language can use the flags parameter only for a few flags, as described below in the section on isql.

The following sections list the parameters for each of the call statement variants.

Call ABF

The statement call abf calls ABF so that you can create or modify the specified application.

application

Specifies tghe application to be worked on. The value is an application name.

flags

Specifies the flags in effect for the application. The value is a list of flags. Includes most flags for the system-level command line (except -u). Separate items in the list with a blank or by pressing the Tab key.

Call Application

The statement call application calls the specified ABF application.

name

> **VMS:** Specifies the name of the image for the application (either a full path name, or the name of a file in the search path).
>
> A symbol defined as a foreign command to run the application.

executable

> Specifies the name of the application (the same as the name parameter). You must specify either name or executable, but not both.
>
> **VMS:** Specifies the full path name of the executable image of the application. You must specify either name or executable, but not both.

frame

> Specifies the *value* is the name of the first frame to be called in the application

constants_file

> Specifies the *value* is the full path and filename, in single quotes, of the file containing global constants values for the application

param

> Equivalent to the -a flag. Lets you pass application-specific parameters to the target application. The value is a string of up to 2000 bytes that contains all parameters to be passed, separated by single spaces.
>
> **VMS:** The value is a string of up to 512 bytes, including up to 256 parameters separated by single spaces.
>
> The parameter list is passed to the target application as a series of parameters to the command that starts the application. The host system that processes this command can modify these parameters before the target application receives them.
>
> Use the CommandLineParameters() function (see page 996) in the target application to refer to the values passed in the parameter list.

To start a second application that uses a different database and you do not want to exit the current application, use the call system statement. This creates two database sessions. Increase your process limits accordingly.

Call Ingmenu

The statement call ingmenu calls the Ingres Menu, the forms-driven interface to all Ingres tools.

flags

Specifies tghe flags to be in effect. The value is a list of flags, including most system-level command line flags (except -u), separated with a blank or by pressing the Tab key.

Call ISQL

The statement call isql calls Interactive SQL, the forms-based Terminal Monitor. For information about using the Terminal Monitor, see the appropriate section of this guide.

flags

Specifies the flags to be in effect. The value is a list of flags, including the following system-level command line flags:

+a | -a
+d | -d
+s | -s

Separate items in the list by typing a blank or by pressing the Tab key. See the description of the isql command in the *Command Reference Guide* for descriptions of the flags.

Call QBF

The statement call qbf calls QBF for use with the specified QBFName, JoinDef, or Table. You cannot pass initial values to QBF. Separate items in the list with a blank or by pressing the Tab key.

qbfname

Equivalent to the -f flag. Runs QBF with the QBFName specified by the *value*. If *value* is the empty string, it starts QBF at the QBFNames catalog frame.

joindef

Equivalent to the -j flag. Runs QBF with the JoinDef specified by the *value*. If *value* is the empty string, it starts QBF at the QBF JoinDefs catalog frame.

querytarget

Places you in the query execution phase and specifies the table, view, JoinDef, or QBFName that you want to query. Follow the querytarget parameter with the name of the object to be queried.

tblfld

Equivalent to the -t flag. Runs QBF with the table specified by the *value*, using a table-field format. If *value* is the empty string, QBF starts at the Tables Catalog frame.

lookup

Equivalent to the -l flag. Runs QBF with the QBFName, JoinDef, or table name specified by the *value*. Ingres looks up the name in this order: QBFName, JoinDef, table. If you do not specify a *value*, QBF prompts you for it.

silent

Equivalent to the -s flag (no value). Suppresses messages. Use an empty string.

mode

Equivalent to the -m flag. The value can be retrieve, append, update, or all. The application user enters QBF in the specified mode. If you do not specify a *value*, QBF prompts you for it.

table

Specifies the name of the table to query with QBF. Do not use with the joindef, qbfname, tblfld, or lookup QBF parameters.

flags

Specifies the flags in effect. The *value* is a list of flags. Includes most system-level command line flags (except -u). Separate items in the list with a blank or tab.

Call RBF

The statement call rbf calls RBF, the Ingres forms-based report facility.

table

Indicates that the *value* is the name of the table or view on which you want to base the report. If you do not specify a report, table, or view name, RBF prompts for it.

mode

Equivalent to the -m flag. Use the mode parameter with the table parameter only; it has no effect with the report parameter.

Optional *values* are: tabular, column, block, labels, indented, or default. If you do not specify a style, RBF selects tabular or block based on the width of the report.

If you do not use the mode parameter, RBF takes one of the following actions:

- If it finds a report, places you in the report specification
- If it finds a table or view, creates a new report specification

mxline

Equivalent to the -l flag. The value is the line length for generating default reports. By default, RBF uses a different line length for each style:

labels = 132 characters

block = 80 characters

wrap = 100 characters

all others = no default line length

The value for mxline, if set, overrides any Report-Writer .pagewidth statement.

report

Equivalent to the -r flag. Indicates that the *value* is a report, rather than a table or view. The *value* is the name of the report. If RBF finds the report, enter the Report Layout frame. If RBF does not find the report, it displays an error message.

If you do not use the report parameter, RBF places you in the report specification if it finds a report, or it creates a new one if it finds a table or view.

silent

Equivalent to the -s flag (no *value*). Use an empty string. Suppresses some informational messages, but not prompts.

flags

Specifies the flags in effect. The *value* is a list of flags. Includes most system-level command line flags (except -u). Separate items in the list with a blank or tab.

Call Report

The statement call report calls the Report-Writer to run the report you specify. Separate items in the list with a blank or by pressing the Tab key.

brkfmt

Equivalent to the +t flag. Causes aggregates to occur over rounded values for any floating point column whose format has been specified in a .format statement as numeric F or template. The *value* is a blank string. Each value is rounded to the precision given in its format, based on displayed data rather than on internal database values.

Syntax:

```
call report(file= 'filename', brkfmt= ' ')
```

file

Equivalent to the -f flag. Directs the formatted report (output) to the file name specified by the *value.* If you do not specify a file name, the report is written to the file specified in output; if you do not specify a filename in either place, the report is displayed on the screen.

forcerep

Equivalent to the -h flag. Creates headers and footers even when the report contains no data, executing all .header and .footer sections and suppressing the detail section. The *value* is a blank string. Allows an .if statement in report footer to verify that no rows were found. See *Character-based Querying and Rporting Tools User Guide* for details.

Syntax:

```
call report(file = 'filename', forcerep = ' ')
```

formfeed

Equivalent to the +b flag. Forces form feeds at page breaks, overriding report-formatting commands. The *value* is a blank string.

Syntax:

```
call report(file = 'filename', formfeed = ' ')
```

mode

Equivalent to the -m flag. Specifies the mode. Values: tabular, wrap, and block. Use the mode parameter with the name parameter only; mode has no effect when used with the report parameter.

If you do not specify a style, Report-Writer creates a default report based on the width of your report. Use pagewidth to change the default.

If you do not use the mode parameter, RBF takes one of the following actions:

- If it finds a report, places you in the report specification
- If it finds a table or view, creates a new report specification

mxquer

Equivalent to the -q flag. Sets the maximum length of the query specified in the .query command to the specified number of characters following all substitutions for runtime parameters. Only needed for very long queries. The *value* is an integer.

mxwrap

Equivalent to the -w flag. Sets the specified number as the maximum number of lines within any block. The *value* is an integer. Default : 300.

name

Names a database table or view for the default report

param

Indicates the parameters for the report. The value is a quoted parameter string or the name of a field containing the parameter string.

Use the format *pname='value'* and separate the string with blanks or tabs (with no blanks around the equals sign.) Enclose nonnumeric values passed in an element in standard quotes, with the following exception.

If *pname* is a character report parameter, enclose value in double quotes within single quotes (' " " '). Place a backslash (\) before embedded double quotes.

VMS: If *pname* is a character report parameter, enclose *value* in a set of double quotes. (Note that it is easier to pass a parameter by calling a report frame.)

nobrkfmt

Equivalent to the -t flag. Causes aggregates to occur over internal database values rather than on the displayed data for any floating point column whose format has been specified in a .format statement as numeric F or template. The *value* is a blank string.

Syntax:

```
call report(file = 'filename', nobrkfmt = ' ')
```

noformfeed

Equivalent to the -b flag. Suppresses form feeds, overriding report-formatting commands. The *value* is a blank string.

pagelength

Equivalent to the -v flag. The *value* is an integer. Sets the page length, overriding .pagelength commands in the report. Default: 61 lines per page if report is written to a file, and your terminal length per page if written to the terminal.

mxline

Equivalent to the -l flag. Sets the page width, overriding .pagewidth commands in the report. Defaults depend on style of report: block: 80; wrap: 100; tabular and column: no default. Needed only for reports with very long lines. The *value* is an integer.

printer

Equivalent to the -o flag. Specifies that the *value* is a printer name (overrides the default printer). Additional memory is required to print large reports.

VMS: Initializes the printer queue to print a job flag page.

copies

Equivalent to the -n flag. Lets you specify the number of copies to be printed. The *value* is an integer. Default: 1 copy.

report

Equivalent to the -r flag. Indicates that the *value* is a report, rather than a table or view, and displays the report. The *value* is a report name.

silent

Equivalent to the -s flag. Suppresses some informational messages, but not prompts. The *value* is a blank string.

Syntax:

```
call report(file = 'filename', silent = ' ')
```

sourcefile

Equivalent to the -i flag. Requests that the report specification be read from a source file outside the Reports Catalog. With this flag, the specified source file is not saved in the database. To save the report specification in the database, use the sreport command. You cannot use this flag in conjunction with the -m or -r flags. You can specify only one filename. Do not specify a report, table, or view on the command line.

-5

Requests report compatibility with Ingres release 5. Use with brkfmt option to ensure compatibility. Displays month of current_date in capitals if no format is specified. With -5 set, the date appears as dd-MMM-yyyy.

-6

Requests that all distinct rows be reported when the following reports are executed: SQL default reports, SQL reports that contain .data, .table, .view, or .sort statements. Without -6, duplicate rows are reported in these SQL reports. Does not apply to QUEL reports.

flags

Specifies the flags in effect. The *value* is a list of flags. Includes most system-level command line flags (except -u). Separate items in the list with a blank or tab.

Call SQL

The statement call sql calls the SQL Terminal Monitor. For more information, see the *SQL Reference Guide.*

flags

Specifies the flags to be in effect. The *value* is a list of flags. Includes the following flags:

+a | -a
+d | -d
+s | -s

Separate items in the list by typing a blank or by pressing the Tab key.

Call Sreport

The statement call sreport calls the Report-Writer to store a report specification in the database.

file

Names a text file containing report-formatting commands for one or more reports

silent

Equivalent to the -s flag (no value). Suppresses status messages. There is no value; use an empty string.

flags

Specifies the flags to be in effect. The value is a list of flags. Includes most system-level command line flags (except -u). Separate items in the list by typing a blank or by pressing the Tab key.

Call VIFRED

The statement call vifred calls VIFRED, the Ingres forms-editing system.

form

Equivalent to the -f flag. Runs VIFRED on the form specified in the value.

table

Equivalent to the -t flag. Runs VIFRED with a default form for the database table specified in the value.

joindef

Equivalent to the -j flag. Runs VIFRED with a default form for the specified JoinDef. The *value* is a JoinDef name.

flags

Specifies the flags to be in effect. The *value* is a list of flags. Includes most system-level command line flags (except -u). Separate items in the list by typing a blank or by pressing the Tab key.

Call Vision

The statement call vision calls Vision so that you can create or modify the specified application.

application

Specifies the application you want to work on. The *value* is an application name.

flags

Specifies the flag(s) to be in effect for the application. The *value* is a list of flags. Includes all system-level command line flags except -u. Separate items in the list by typing a blank or by pressing the Tab key.

Examples

Call the Inventory application, beginning at the Parts frame:

```
call application (name = 'inventory',
  frame = 'parts');
```

Call the application specified in the Appname field of the current form, starting at the frame in the Framename field:

```
call application (name = appname,
  frame = framename);
```

Call QBF with no parameters. The QBF catalog frame is displayed:

```
call qbf ();
```

Run QBF in the append mode, suppressing status messages, using the QBFName "expenses":

```
commit;
call qbf (qbfname = 'expenses',
  flags = '-mappend -s');
```

Run a default report on Emptable in the tabular mode:

```
commit;
call report (name = 'emptable',
  mode = 'tabular');
```

Call report on a temporary table. Both temporary table's name and contents are based on user input:

```
create table :tbl as select * from orders
  where qualification (status = status,
    custno = custno);
commit;
call report (name = 'orders',
  param = 'table="' + :tbl + '"');
```

The following example illustrates how to pass different data types in a **call report** statement:

- Use the following Report-Writer query specification:

```
.QUERY select * from callreptable
  where db_int = $rw_int
    and db_char = '$rw_char'
    and db_date = '$rw_date'
```

- Include the following 4GL code in your application:

```
initialize (h_param = c80 not null) =
begin
end

'Callrep' =
begin

/* Construct a parameter string by */
/* concatenating fields or variables with */
/* text constants. Data comes from fields */
/* "scr_int," "scr_char," and "scr_date," */
/* with data types INTEGER, CHAR, AND DATE, */
/* respectively. As they are concatenated, */
/* non-character fields are converted to */
/* character type using the function VARCHAR. */

h_param = 'rw_int=' + varchar(:scr_int)
           + ' rw_char=\"' + :scr_char + '\"'
           + ' rw_date=\"' + varchar(:scr_date) + '\"';

call report (report = 'callrep',
              param = :h_param);

end
```

Note: Character and date values can be delimited by single quotes as follows, instead of by backslashes and double quotes as above:

```
h_param = 'rw_int=' + varchar(:scr_int)
           + ' rw_char=''' + :scr_char + ''''
           + ' rw_date=''' + varchar(:scr_date) + '''';
```

Call System

Calls the operating system from within an application.

Syntax

```
call system [systemcommand]
```

systemcommand

Specifies a string expression corresponding to an external system command

Description

The call system statement calls the operating system and executes a single specified *systemcommand* before returning automatically to the application. Use the call system statement without specifying a command to fork and execute a shell process and place you at the operating system prompt.

VMS: Use the call system statement without specifying a command to spawn a DCL sub process and place you at the operating system prompt.

You can then execute any operating system commands and return to the application by typing the logout command.

The call system statement automatically clears the frame before calling the operating system. When control returns to the application, execution resumes with the statement immediately following call system. The current form is redisplayed at the end of the activation block. The call system command cannot return status information to the application.

When you are at the operating-system level after issuing a call system statement, do not change your default directory. This can cause subsequent call statements to fail.

Note: While you can use call system to start a second ABF or Vision application, it is more efficient to use the call application statement. See Call (see page 949) for details.

VMS: The VMS inquire command does not execute properly in a command file called with call system. For an alternative, see Using the Call System Statement (see page 899).

Examples

Call the operating system to execute the command specified in the Cmdname field on the current form:

```
call system cmdname;
```

Windows: Call the operating system to edit the file *test.txt*.

UNIX:

```
call system 'ed test.txt';
```

or

```
filename = test.txt;
call system = 'ed ' + :filename;
```

VMS:

```
call system 'edit test.txt';
```

or

```
filename = test.txt;
call system = 'edit ' + :filename;
```

UNIX:

Call the operating system to execute a shell script:

```
call system 'sh test.sh';
```

VMS:

Call the operating system to execute a command procedure:

```
call system '@test.com';
```

Callframe

Transfers control from the current frame to another ABF frame.

Syntax

```
[returnfield :=] callframe calledframe
  [([parameterlist])] [with style = fullscreen
  [(screenwidth = value)] | popup [(optionlist)]]
```

returnfield

Specifies the field in the calling frame or procedure that receives the return value from the called frame. The data type of the return field must agree with the return type of the called frame as declared in ABF.

calledframe

Specifies the ABF name of the frame you are calling (a 4GL name)

parameterlist

A series of assignment statements separated by a semicolon (;) or comma (,). Do not use a comma where it results in ambiguity, such as following a parameter that is a query with a name list as its last component (such as from or order by clauses). Use (;) to separate query parameters from other parameters in the list.

Each assignment statement links a field or a variable in the initialize section in the called frame with an expression, as shown below.

Simple assignments:

```
simpleobjectname  := expression
```

or

```
simpleobjectname := byref (simpleobjectname)
```

On the left hand side of the assignment: *simpleobjectname* is the name of a field in the called form or a simple variable in the initialize section of the called frame.

On the right hand side of the assignment: *simpleobjectname* is the name of a field in the calling form or a simple variable in the calling frame.

Expression is any legal 4GL expression. Usually it contains values from the calling frame or from a database table. The data types of the expression and the simpleobjectname must be compatible.

For a simple object, byref() indicates that the object is passed by reference; this is discussed below. A complex object is always passed by reference; do not include the byref() statement.

Complex assignments:

```
arrayname := arrayname

recordname  := recordname

tablefieldname  := database query

calledformname  := database query

calledformname := callingformname.all

calledformname  := tablefieldname[[integerexpr]].all
```

On the left hand side of the assignment: *arrayname and recordname* are the names of objects in the initialize section of the called frame. *Tablefieldname* is the name of a table field in the called frame.

On the right hand side of the assignment: *arrayname* and *recordname* are the names of objects in the calling frame. *Tablefieldname* is the name of a table field in the calling frame.

The parameter list can include assignment statements for any mix of form object types. However, 4GL allows only one database query within the parameter list.

screenwidth = *value*

Specifies the screen width for a fullscreen frame. The *value* is a keyword, integer constant, or integer variable as shown:

0 | default

Specifies VIFRED definition

1 | current

Specifies current screen width

2 | narrow

Specifies narrow width for screen, typically 80

3 | wide

Specifies wide width for screen, typically 132

intvar

Specifies a 4GL expression evaluating to 0, 1, 2, or 3

optionlist

Specifies a list of style options for pop-up position and border style. Syntax:

```
option = value [,option = value]
```

Options: startcolumn, startrow, border.

startcolumn and startrow specify the coordinates for the upper left corner of the pop-up form. The *value* is a keyword, integer constant, or integer variable as follows:

default | 0

Specifies VIFRED definition or default

floating | -1

Specifies current cursor position

int

Specifies a positive integer indicating the number of rows or columns from the origin. The origin (1,1) is the upper left corner of the screen.

intvar

Specifies a 4GL expression evaluating to -1, 0 or a positive integer

border

Specifies the form's border type. The *value* is a keyword, integer constant, or integer variable as follows:

default | 0

Specifies VIFRED definition or default

none | 1

Specifies no border

single | 2

Specifies single-line box border

intvar

Specifies a 4GL expression evaluating to 0, 1, or 2

Description

The 4GL callframe statement is the only means by which an operation, such as a menu operation, can transfer control to, or call, a new frame. The calling operation appears in the *calling frame*, and the frame to which it transfers control is known as the *called frame*.

You can include an optional with style clause to override the called frame's original style definition (as specified through the Application-By-Form's FormEdit operation). The two available styles are popup or fullscreen. You can specify options to modify pop-up or full-screen behavior.

Popup

You can use variables in the *optionlist* to specify the pop-up form's exact location dynamically at run time. If there is not enough room in the window to start the pop-up form at the user-specified location, the forms system displays the form in fullscreen width.

Fullscreen

This option forces the FRS to use the entire window to display the form. You have four choices for the *screenwidth* option: current, default, narrow, and wide. Use VIFRED to specify a width for a form. You can override these settings at run time.

You can nest called frames so that the called frame calls another frame, and so on. A frame cannot call any frame whose form is associated with a frame at an outer level of nesting.

You can pass the value in derived fields or columns to another frame in a callframe statement, but you cannot pass a value to a derived field with any statement.

You can use a return statement in either of two ways: to exit a called frame or to pass a value back to the calling frame, as follows:

- To exit a called frame*,* you normally use a return statement to transfer control back to the calling frame.
- To pass a value back to the calling frame*,* specify an expression on the return statement in the called frame. In the calling frame, place a *returnfield* on the left side of an assignment statement, with the entire callframe statement on the right side.

The value returned must meet the following requirements:

- The value can be of any data type, as long as you have previously declared it in the ABF User-Specified Frame Definition frame for the called frame.
- The value must be compatible in data type and nullability with the return field.

Under certain circumstances, you can return additional values to a calling frame by means of the byref option.

For information on returning a value to the top frame, see Returning to the Top Frame (see page 1337).

The callframe statement can pass values to the called frame by means of an optional *parameter list*. These values are supplied either by fields in the calling frame or by a query against the database.

Using different versions of the *parameterlist*, you can pass values to a form, a table field, simple fields, a record, or an array in the called frame.

The with style options in optionlist or screenwidth override the definition of the form made through the ABF FormEdit operation. If startcolumn and startrow specify a numeric location for the pop-up form, the called form appears at that specified location, regardless of how the form is declared (that is, fullscreen, popup/floating, or popup/fixed).

For example, when your application calls a form declared as a fixed-location pop-up with the startcolumn and startrow set to "floating" or "-1," the form appears at the "floating" location (the current cursor position) rather than at the fixed location.

The forms system attempts to display floating pop-up forms without obscuring the field that the cursor occupies. If there is not enough room on the frame to display it at the specified location, an error message appears and the forms system displays the form as close as possible. If the window is too small to display the pop-up form and its border, the form appears in fullscreen mode.

Passing Values from the Calling Frame

You can pass values from the form of the *calling frame* to the form of the *called frame* by means of a series of assignment statements in the callframe parameter list. You can pass values to a specified simple field, to all simple fields, or from a table-field row to the corresponding simple fields in a form, or to arrays. Each of these methods is discussed below.

To a Simple Field from the Calling Frame

You can pass a single value to a simple field or local variable in the called frame using the syntax shown below. This is like a simple assignment. You can optionally prefix the form name of the called frame to the *fieldname* that is to receive the value:

```
[calledformname.]fieldname := expression
```

You can also pass a value to a called frame with the byref statement. This allows the field in the calling frame that contains the passed value to be changed by the processing in the called frame. See Passing Parameters by Value or by Reference in Using the Callframe Statement (see page 889) for further discussion of passing values by reference.

Use this syntax:

```
[calledformname.]fieldname := byref (formobject)
```

Formobject is a field name, a table-field column name, or a local variable in the calling frame.

To All Simple Fields from the Calling Frame

Use the option all to assign the values in all the simple fields in the calling frame to simple fields in the called frame. To do so, use this syntax for the *parameterlist*:

```
calledformname := callingformname.all
```

In this case, each simple field in the calling frame must correspond in name and data type to a field in the called frame. The called frame can, however, contain fields not corresponding to fields in the calling frame. The mapping ignores any local variables.

To Simple Fields from a Table-Field Row

Use the option all to assign the values in all the columns of a table-field row in the calling frame to simple fields in the called frame, using this syntax:

```
calledformname := callingtablefieldname
    [[integerexpr]].all
```

Each table-field column in the calling frame's form must correspond in name and data type to a field in the form of the called frame. However, the called frame can contain additional fields that do not correspond to fields in the calling frame.

The row from which the values are passed is the row on which the cursor is currently resting, unless you specify another row by its number in an integer expression (enclosed in square brackets or vertical bars). In this case, the values are taken from the row with that number. The mapping ignores hidden columns.

Passing a Query

4GL allows a query to be passed as a parameter to the called form or to a table field in the called form, utilizing the select statement. For syntax and parameters, see Select to a Complex Object (see page 1111).

A query passed as a parameter to the called form is similar to an attached query, except that the query submenu is the activation list for the called frame. When the query is executed, it runs to completion, and the selected rows are read into a temporary file. Each next statement in the called frame causes the next row from the temporary file to be displayed.

You can use the *qualification* function in a where clause or enclose the entire where clause in a variable.

The section on the select statement contains more detailed information on writing queries. However, some special rules apply for queries to be passed as parameters.

To write the results of a query to a table field, include both the form and the table-field name, as in the example:

```
callframe newframe (newform.personnel :=
  select …)
```

Also, you must place the names of fields in the called frame on the left side of the target list, while names of fields in the calling frame must appear on the right in the where clause. For example, consider the callframe statement:

```
callframe newframe (newform := select
  empnum = empnum from projemps
  where projemps.projnum = :projnum)
```

Here *empnum* is the name of a field in the called frame and therefore appears on the left side of the target list, while *projnum* is a field in the calling frame and appears on the right in the where clause.

Important! If fields in the left-hand side do not exist in the called frame, 4GL does not generate errors until run time.

Queries passed to another frame, like any other query, are always part of a transaction. Shared locks remain on the selected data until the next commit or rollback statement. You can override this default and release locks on the selected data by one of the following methods:

- If your application has previously issued a set autocommit on, then a commit automatically takes place after every successfully executed query.
- If you have turned off read locks on the table from which you are selecting via the set lockmode statement, then locks are not held on the table from which you selected. However, a transaction is still in progress.

See the *SQL Reference Guide* for detailed descriptions of the set lockmode statement.

Examples

Call Newframe with style pop-up, passing a value from the Projnum simple field of the calling frame to the Projnum simple field in the called frame. Because the value in the calling frame is not passed by reference, it is not affected by changes made to the corresponding field in the called frame.

```
Callframe newframe (newform.projnum :=
    projnum)
  with style=popup;
```

Call Newframe, passing a value from the Employee simple field in the calling frame to the Employee simple field in the called frame. Because the value in the calling frame is passed by reference, it is affected by changes made to the corresponding field in the called frame.

```
Callframe newframe (newform.employee :=
  byref (employee));
```

Call Newframe, passing values from all columns in the current row of the Oldtable table field to corresponding simple fields in Newform:

```
callframe newframe (newform :=
  oldtable.all);
```

Call Newframe and pass values into the Projnum and Name simple fields in Newform from the database:

```
callframe newframe (newform := select
  projnum, name = projname
  from projinfo
  where projnum = :projnum);
```

Call Newframe, passing values into the Projnum simple field and the Personnel table field of Newform. The example retrieves information from the Projemps table concerning all employees assigned to a particular project. The Personnel table field has at least the columns Empnum and Hours.

```
Callframe newframe (newform.projnum :=   projnum;
  newform.personnel := select   empnum,
  hours = emphours
  from projemps
  where projnum = :projnum);
```

Call Newframe, placing a value in the Status field upon returning to the calling frame:

```
status := callframe newframe;
```

Call a report frame, passing in parameters:

```
deptno = 2;
lname = 'Smith';
callframe reportframe (name = lname;
  deptno = deptno);
```

Call Newframe, passing the name attribute of the third record of Emparray to the corresponding simple field:

```
callframe newframe (newform.name = emparray[3].name);
```

Call Newframe in fullscreen mode, using the wide screenwidth:

```
callframe newframe with style = fullscreen
  (screenwidth = wide);
```

Callproc

Calls a database procedure, a procedure written in 4GL, or a procedure written in a supported 3GL such as ADA, C, COBOL, FORTRAN. Your system cannot support all of the programming languages listed.

Syntax

For database procedures or procedures written in 4GL:

```
[returnfield = [callproc] ] procname
 [([parameterlist])]
```

For procedures written in high-level 3GLs:

```
[returnfield = [callproc] ] procname
  [([expression | byref (expression)
  {, expression | byref(expression)}])]
```

returnfield

Specifies the name of a field in the calling frame or procedure to receive the return value from the called procedure. The data type of the return field must agree with the return type of the called procedure.

procname

Specifies the name of the procedure you are calling. Enclose procedure names that are the same as 4GL keywords in quotes.

You cannot specify the name of a local 4GL procedure as the value of a variable.

parameterlist

Specifies a series of assignment statements separated by a semicolon (;) or comma (,).

Each assignment statement links a field or a variable in the keyword parameters of the initialize section in the called procedure with an expression, as shown below.

Simple assignments:

```
simpleobjectname :=  expression
simpleobjectname := byref(simpleobjectname)
```

On the left hand side of the assignment: *simpleobjectname* is the name of a field in the called procedure or a simple variable in the initialize section of the called frame.

On the right hand side of the assignment: *simpleobjectname* is the name of a field in the calling procedure or a simple variable in the calling frame.

For a simpleobject, byref() indicates that the object is passed by reference; this is discussed below. A complex object is always passed by reference; do not include the byref() statement.

Complex assignments:

```
arrayname := arrayname
recordname := recordname
callingformname.all
tablefieldname [[integerexpr]].all
```

On the left hand side of the assignment: *arrayname* and *recordname* are the names of objects in the initialize section of the called procedure.

On the right hand side of the assignment: *arrayname* and *recordname* are the names of objects in the calling procedure*.*

The .all syntax maps the simple fields in the calling form (or columns in the named table field) to the parameter list inside the procedure. Any field or column that does not correspond to a parameter is ignored.

expression

Specifies any legal 4GL expression that produces a simple-typed value

byref

Specifies a keyword indicating that a variable must be passed to the called procedure *by reference*

Description

The 4GL callproc statement calls a procedure from an ABF frame or procedure. A procedure is a routine written in 4GL, SQL, or a 3GL. A 4GL procedure can be global or local to its source file.

When using procedures, you can:

- Include embedded query language statements in a procedure
- Specify a procedure that returns a single value within an expression to the calling frame
- Return multiple values to the calling frame by reference through the procedure's parameter list
- Pass simple data types as parameters

You can use the reserved word all in the parameter list to a 4GL procedure, as shown below:

```
callproc procname (calling_formname.all |
  calling_tablefield_name[[integer_expr]].all)
```

This syntax maps the simple fields in the calling form (or columns in the named table field) to the parameter list inside the procedure. Any field or column that does not correspond to a parameter is ignored.

Local variables and hidden columns are not mapped. When all is used with a table-field name, values are taken from the row on which the cursor is currently resting. To override this, specify an integer expression enclosed by brackets following the table-field name. In this case values are taken from the visible row with that number.

The keyword callproc is necessary except when the procedure returns a value that is used as a complete expression or as a constituent of an expression. In the latter case, you must omit the keyword callproc.

If you omit the keyword callproc, and you are not passing any parameters to the procedure, you must specify a pair of parentheses following the procedure name; for example:

```
x = proc1 ();
```

The name of the procedure must be the same as the one you give it within its source code. If your procedure name conflicts with a system-defined function, your procedure overrides the system-defined function, except when you use the system-defined function within a query statement. You cannot call 4GL procedures within query statements. Use the Appplication-By-Forms -w flag to detect conflicts.

See your query language reference guide for a list of system-defined functions and further information on the -w flag.

When a parameter is passed to a procedure, the parameter's value is transferred to a corresponding parameter inside the procedure. A parameter that is a simple field or a table-field column can be passed either by value or by naming it in the statement.

The value for a field or table-field column passed by reference is not updated until the procedure returns to its calling frame.

When passing the value of a field or a column to a procedure by reference, be sure its data type, including nullability, is compatible with the corresponding parameter in the procedure. 4GL checks this at run time for 4GL procedures, but cannot check the data types for procedures written in other languages.

Because procname is a 4GL name, you can specify the name of the procedure at run time (unless the procedure is a local procedure). In this case, 4GL assumes that the return type is the same as the result value type. If the procedure attempts to return a value of an incompatible type, an error results at run time.

The following example calls the procedure specified in the *procfield* field of the current form and returns a value to the *result* field:

```
result = :procfield (x = 5, y = price);
```

Because the procedure name is a 4GL name, the colon is required when the value is to be supplied at run time. At run time, the application code supplies the correct procedure name for *procfield* on the current form, or it can be entered by you; then the named procedure is called. Note that the value of *procfield* cannot be a local procedure.

The following sections discuss 4GL procedures, 3GL procedures, and database procedures. The following 3GL procedures each have a separate section: ADA, C, COBOL, FORTRAN. Your system cannot support all of the programming languages listed. 4GL procedures are also discussed in Writing 4GL Statements (see page 849).

Coding 4GL Procedures

Use the procedure statement when coding a 4GL procedure in a source-code file.

```
procedure identifier [( [ variable =typedeclaration
  {,  variable = typedeclaration} ] ) ]=
[ declare variable = typedeclaration
  {, variable = typedeclaration} ]
[begin | {
    statement; {statement;}
end | }]
```

The variables listed in parentheses after the procedure name (*identifier*) are keyword parameters; you can place values in them when the procedure is called by another frame or procedure. The 4GL procedure statement recognizes record types in variable declarations.

You can also declare local variables. To do this, use the keyword declare following the procedure header. Variables defined in this declare section cannot have values passed to them through the call parameter lists either positionally or by keyword. Their scope is local; limited to the procedure in which they are defined.

All form fields and table field columns are accessible as keyword parameters for 4GL frames and procedures. To declare a data type, use this syntax:

```
simpletype [(length[,scale])] [with|not null];
recordtype;
array of recordtype;
array of type of table tablename;
array of type of form formname;
array of type of tablefield form.tablefield;
```

Any parameter in the 4GL procedure that is not passed a value receives a default value of either "0" or an empty string, according to its data type. However, any parameter specified in the callproc statement must exist as named in the procedure being called. (Parameter names are 4GL names.)

A 4GL procedure can be global or local to its source file, as discussed in the following subsections.

Using Local 4GL Procedures

A local 4GL procedure is a series of 4GL source statements that appear in the source file for a user-defined frame or a global procedure. The ABF Frame Catalog has no record of local procedures.

You must declare *and* define each local procedure in the source file. The declaration specifies the procedure's return type; the definition specifies the actions that the procedure performs.

You must declare local 4GL procedures in one of the following places:

- The declare section of the global procedure (if the source file is for a global procedure)
- The declare section of the initialize statement (if the source file is for a frame)

You can mix procedure and variable declarations within these locations. Each local procedure declaration takes one of the following forms:

identifier* = procedure returning *typedeclaration

This syntax indicates the type of the value the procedure is expected to return.

Procedures can return simple data types (for example, integer or varchar) only; they cannot return complex data types (such as arrays).

***identifier* = procedure returning none**

or

***identifier* = procedure**

These two forms are equivalent and indicate that the local procedure returns no value.

You must define local 4GL procedures at the end of the source file, as follows:

- If the source file defines a global procedure, the local procedure definitions must follow the definition of the global procedure.
- If the source file defines a frame, the local procedure definitions must follow all the activations for the frame.

Each local procedure definition consists of a procedure statement, using the syntax shown at the beginning of this section.

Note: Procedure names are case sensitive. Be sure the call statement, declaration, and definition use exactly the same name.

Local 4GL procedures:

- Cannot use forms statements if the local procedure is inside of a 4GL global procedure.
- Can use some forms statements if the local procedure is inside of a 4GL frame. The following forms statements, with some restrictions, can be used:
  - clear, redisplay, validate, and validrow are unrestricted
  - resume must be inside of a submenu (either run submenu or display submenu) loop
  - resume field, resume next, resume entry, and resume menu must be inside of a display submenu loop
- Cannot declare hidden columns in table fields.
- Cannot be called from a different source file.
- Can refer to fields or local variables declared in the frame or global procedure at the beginning of the source file, except when local variables of the same name are declared in the local procedure.

  For example:

  ```
  procedure my_global_proc =
  declare
     x = integer,
     y = integer,
     my_local_proc = procedure
  begin
  . . .
  end
  procedure my_local_proc =
  declare
    y = integer
  begin
    . . .
  end
  ```

  A reference to *x* within the body of "my_local_proc" refers to the *x* declared in "my_global_proc." However, a reference to *y* within the body of "my_local_proc" refers to the *y* declared in "my_local_proc;" the local procedure cannot refer to the *y* declared in the global procedure.
- Can use "*" or all in various statements (such as select, insert, and update) to refer to all visible fields in a form or table field.

  Such references always retrieve all visible fields, even if the local procedure defines a variable with the same name as one of the visible fields.

Using Global 4GL Procedures

You must declare a global 4GL procedure on the ABF Create a Procedure frame, specifying the name of a source file that contains the procedure. The 4GL code that defines the procedure must appear at the beginning of the source file that you specify.

The fields, table fields, form, and variables that you declare in a global procedure are available to local procedures. The only exception is when a local procedure defines a variable of the same name as used in a global procedure.

Global 4GL procedures do not use forms. Therefore, the following 4GL forms-control statements related to form display and table fields—clear, redisplay, resume, validate, and validrow—are not available within a global 4GL procedure or within a local 4GL procedure in the same file. Also, the following statements are available with arrays only; you cannot use them with table fields: clearrow, deleterow, insertrow, and unloadtable.

3GL Arguments

The following table summarizes how to declare, within your 3GL procedure, parameters to pass from your 4GL application to your 3GL procedure:

4GL Data Type	3GL Variable Declaration
varchar, text	A variable-length character string. Unlike the fixed-length character types char and c, these variable-length character fields are not extended with blank characters before being passed to 3GL; only the amount of data entered into the field is passed to 3GL. Your 4GL program can determine the declared size of a field and the current amount of data in the field by calling the size and length string functions.
char, c	A fixed-length character string. The fixed size is determined by the length of the field on the form as declared using the ABF FormEdit operation, or, for a global variable, by its declared size in 4GL. Data entered into the field is padded with blank characters up to its full declared length before being passed to 3GL.
date	Fixed-length, 25-byte character string.
money	Extended precision (8-byte) floating point.
integer, I	Integer.
decimal	A fixed-point exact number. Depending on the 3GL, a decimal value is declared as either a packed decimal

4GL Data Type	3GL Variable Declaration
	or a float.
float, f	Floating point.

All data types can also be passed to 3GL with the byref option. When working in 3GL with character data passed to it by reference, be careful not to extend a character variable beyond its 4GL declared length.

You cannot pass a char or varchar variable that contains an embedded zero byte (hexadecimal '00') to 3GL. A truncated version of the 4GL variable can be passed.

When a 4GL date variable passes to a 3GL routine by reference, or when the return type of a 3GL procedure is date, then 4GL receives it back from the 3GL routine as a 25-byte string. You must then convert it to Ingres internal date format. The date must be valid before the conversion can succeed.

3GL procedures do not support Null values passed from 4GL. Passing a 4GL nullable-type variable containing a Null value causes a runtime error. Use the IFNULL function and IS NULL comparison operator to pass nullable types between 3GL and 4GL. For example:

```
IFNULL (v1, v2)
```

This reference returns the value of *v1* if *v1* is NOT NULL; otherwise, if *v1* is NULL, it returns the value of *v2*. The variables *v1* and *v2* must be of the same data type. (See your query language reference guide for more information on IFNULL.)

If an impossible value exists for the argument, use the impossible value to indicate a NULL:

```
callproc empcalc (IFNULL (age, -1));
```

Coding ADA Procedures (VMS only)

To call an ADA procedure from 4GL, use the following steps:

1. Create the procedure from the operating system as described below.
2. Declare the procedure to ABF as a library procedure.
3. Declare the link options file to ABF in the Defaults window.
4. Access ABF and run the application.

If there are changes to be made to the procedure, you have to compile it at the operating system level using 'ada *adaproc*', where *adaproc* is the name of the procedure.

Create an ADA Procedure

To create an ADA procedure

1. At the operating system level, create a file containing the procedure. For example, the following procedure is in a file called adaproc.ada.

   ```
   with system, lib, text_io;
   use text_io;
   function adaproc (int_val : in integer;
     float_val : in long_float) return integer is
     ret_val : integer;
   begin
     ret_val := int_val;
     return ret_val;
   end adaproc;
   pragma export_function(adaproc);
   ```

2. Create and run a command file that creates the library directory, compiles the procedure and stores the object file in the library directory.

   For example, the following procedure is named adaproc.

   ```
   $ acs create library [.adalib]
   $ acs set library [.adalib]
   $ acs set pragma/long_float=d_float
   $ ada adaproc.ada
   $ acs export adaproc
   ```

3. Create a linkoptions file that points to the object code in the library directory. For example, put the following line in a file called linkfile.opt:

   ```
     $2$DUA18:[directory_path.ADALIB]adaproc.obj
   ```

Coding C Procedures

Use the following format when coding a C procedure:

```
procname([parameters])
{
  processing statements
}
```

In C, the procedure must not be named "main," and it must not be static. You can call any other C procedure from 4GL.

In C, observe these passing modes:

- Pass an *integer* as four bytes by value (or by pointer, if byref is specified).
- Pass a *float* as a double-format float by value (or by pointer, if byref is specified).
- Pass a *string* as a pointer to a null-terminated string.
- Pass *fixed-length* string types (c, char) with trailing blanks up to their full length.
- Pass *variable-length* string types (text, varchar) without trailing blanks.

Consider this call to a procedure *q*:

```
callproc q (1 + 2, 2.3, 'This is a string');
```

The following declarations are required:

```
q(x, y, z)
long x;
double y;
char *z;
{
  processing statements
}
```

To pass *x* and *y* by reference, make the following changes to their formal argument declarations:

```
long *x;
double *y;
```

No changes are necessary to pass *z* by reference.

A C procedure must return an int, a double, or a char ***** value, as shown below.

To return an *integer:*

```
int
reti()
{
  return 10;
}
```

To return a *floating-point value:*

```
double
retf()
{
  return 10.5;
}
```

To return a *character string:*

```
char *
rets()
{
  return "Returned from rets";
}
```

Any C procedure that returns a char ***** value to 4GL must return a pointer to a valid address (a string constant or static or global character buffer). The procedure cannot return a pointer to a local character buffer.

Coding COBOL Procedures

Use the format below when coding a COBOL procedure:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Procname.
ENVIRONMENT DIVISION.
DATA DIVISION.
  Processing statements
LINKAGE SECTION.
  Processing statements
PROCEDURE DIVISION
  processing statements
```

In COBOL, you pass integers as four bytes by reference (by name), floats are passed as double format floats by name and characters are passed by name, as the address of the character argument. You can pass decimals as double format floats by name, or as packed decimals. Specify whether you want to pass decimals as floats or decimals on the ABF or Vision procedure definition frame. Passing decimal values as floats allows compatibility with previous versions of Ingres, which did not support the decimal data type.

The length for a character string, as specified in the COBOL procedure, must be the length of the actual argument. In the example below, *z* is given a PICTURE of *X(16)* because the actual argument ('This is a string') is 16 characters long. In general, COBOL only has fixed-length strings. When passing string values from 4GL to COBOL you must use one of the fixed-length types, c or char. This guarantees that a blank-padded string of a known length is passed.

For example, in the initialize section of a frame, declare the local variable *to_cobol* with the following format:

```
to_cobol = char(16)
```

Consider this call to a procedure *q*:

```
to_cobol = 'This is a string';
callproc q(1+2, 2.3, to_cobol);
```

The following declarations are required:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Q.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 x PICTURE S9(8) USAGE COMP.
01 y USAGE COMP-2.
01 z PICTURE X(16).
PROCEDURE DIVISION USING x,y,z
```

In the previous declarations, the decimal values for decimal literals as well as variables are passed as floats. To declare the decimal value as a packed decimal, use declarations similar to the following for a 4GL decimal(5,2):

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Q.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 x PICTURE S9(8) USAGE COMP.
01 y PICTURE S9(3) V9(2) USAGE COMP-3.
01 z PICTURE X(16).
PROCEDURE DIVISION USING x,y,z
```

In COBOL, you can return an integer value only, as shown below:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Reti.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 r PICTURE S9(8) USAGE COMP.
LINKAGE SECTION.
PROCEDURE DIVISION GIVING r.
sbegin.
  MOVE 10 TO r.
EXIT PROGRAM reti.
```

Note that the PICTURE S9(8) data type in COBOL corresponds to the 4-byte integer data type in 4GL.

Coding FORTRAN Procedures

Use the format described for your operating system.

Coding FORTRAN Procedures (UNIX)

Use the format below when coding a FORTRAN procedure:

```
subroutine procname
    processing statements
end
```

In FORTRAN, the procedure is declared as a subroutine or function. It cannot be a program.

Integers are passed as four bytes by reference, floats are passed as double-format floats by reference, and strings are passed in accordance with the string format used by your FORTRAN compiler. Use the character ***(***size***)** format for all character types, where *size* corresponds to the actual length of the table field or is large enough to hold any expected string. Consider this call to a procedure *q*:

```
callproc q (1 + 2, 2.3, 'This is a string');
```

The following declarations are required:

```
subroutine         q(x,y,z)
integer            x
real*8             y
parameter          (MAXSTR = 512)
character          *(MAXSTR)z
```

FORTRAN subroutines cannot return values to 4GL. To return a status to 4GL, the called procedure must be a FORTRAN function rather than a subroutine.

A FORTRAN function returns integer, real*8, or character ***(***size***)** values. The maximum allowable size for a string is 512 bytes. FORTRAN does not return the actual size of the string to 4GL. If the receiving string in 4GL is a variable-length character type, trailing blanks are truncated when the string returns. The three procedures are shown below.

To return an *integer:*

```
integer function reti
reti = 10
end
```

To return a *floating-point value:*

```
real*8 function retf
retf = 10.5
end
```

To return a *character string*:

```
character *(20) function rets
rets = 'Returned from rets'
end
```

Coding FORTRAN Procedures (VMS)

Use the format below when coding a FORTRAN procedure:

```
subroutine procname
    processing statements
end
```

In FORTRAN, the procedure is declared as a subroutine or function. It cannot be a main program.

Integers are passed as four bytes by reference, floats are passed as double-format floats by reference, and strings are passed by descriptor. Use the character *(*) format for varchar and vchar types. The fixed-length format character (*size)* can be used for 4GL char and c data types.

Consider this call to a procedure *q*:

```
callproc q (1 + 2, 2.3, 'This is a string');
```

The following declarations are required:

```
subroutine        q(x,y,z)
integer           x
real*8            y
character *(*)    z
```

FORTRAN subroutines cannot return values back to 4GL. To return a status to 4GL, the called procedure must be a FORTRAN function rather than a subroutine.

A FORTRAN function must return integer, real*8, or character*(*) values. The maximum allowable size for a string is 512 bytes. FORTRAN does not return the actual size of the string to 4GL. If the receiving string in 4GL is a variable-length character type, trailing blanks are truncated when the string returns. The three procedures are shown below.

To return an integer:

```
integer function reti
reti = 10
end
```

To return a floating-point value:

```
real*8 function retf
retf = 10.5
end
```

To return a character string:

```
character *(*) function rets
rets = 'Returned from rets'
end
```

Calling Database Procedures

Database procedures are composed of SQL statements and are stored as objects in the database. You use the SQL create procedure statement to create database procedures outside of ABF.

The following SQL code creates the database procedure **tax_emp()**:

```
create procedure tax_emp(
  employee_id integer not null,
  tax_percent integer not null)
as
begin
  /* processing statements */
end;
```

You then can call this procedure from within an application with the following 4GL statement:

```
retval = tax_emp (employee_id, 5);
```

The following rules apply to database procedures:

- Database procedures cannot contain data definition statements, such as create table.
- Database procedures cannot create or drop other procedures.
- Database procedures cannot call other procedures or frames.
- You cannot use a repeated clause in a database procedure; the procedure itself provides the same benefits as the repeated clause.
- A select statement in a database procedure must assign its results to local variables; if the statement returns more than one row of data, only the first row is assigned to the result variable.
- You can use procedure parameters or local variables in place of constant values in a database procedure.
- Database procedures cannot contain any statements that refer to forms.

See the description of the create procedure statement in the *SQL Reference Guide* for a more detailed discussion of database procedures, including a list of SQL statements that they can contain.

Calling Library Procedures

To call a procedure for which you have the object code but no source code, perform the following steps to link the procedure into an application:

1. Specify the link options filename in the Applications Defaults window, or with the ING_ABFOPT1 logical or environment variable. The filename specified in the Application Defaults window overrides the value specified by the ING_ABFOPT1 logical or environment variable.
2. List the object file for the procedure (or the library that contains the object code) in the link options file.
3. Define a library procedure to ABF. Use the same name for the procedure as you use in the callproc statement. Set the symbol to refer to the procedure. The symbol name can differ from the procedure name.

The reference is resolved later in the link list using the object files and libraries specified by the default link options file.

ADA procedures are always library procedures.

Examples

This example calls the 4GL procedure *addtax*, passing the *taxpercent* field by value and the *price* field by reference. Thus, a computed value can be returned in the *price* field:

```
callproc addtax (tax = taxpercent,
  total = byref(price));
```

The following example checks whether the global variables in an application have been set. If not, it calls the procedure Set_globals to set them. Create the integer global variable, *globals_set,* in ABF.

```
if (globals_set = 0), then
    /* indicates that globals are not set */
  callproc set_globals();
    /* set_globals sets the values */
    /* for the global variables */
  globals_set = 1:
    /* indicates that globals are now set */
endif;
```

Clear

Clears information from the screen.

Syntax

```
clear field fieldname {, fieldname}

clear field all

clear screen
```

Description

The 4GL clear statement has three variants, as shown above.

clear field *fieldname* {, *fieldname*}

This statement clears one or more fields on the currently displayed form, initializing them to blank, zero, or null values depending on the nullability of each field.

The *fieldname*, a 4GL name, can be either the name of a simple field or the name of a table field; it cannot refer to a local variable. If *fieldname* names a table field, all the rows of the table field are cleared and the data set is deleted. Use mode *update* to restore cleared field values. Set_forms form (mode = *update*) does not restore old values.

You cannot directly clear a derived field. However, clearing one of its source fields has the effect of clearing the derived field.

If clear field *fieldname* or clear field all is embedded in an operation, such as a menu operation, the fields on the form are cleared at the end of the operation. The fields are not cleared until all the other statements in the operation are completed, and control returns to the calling screen. If you want to clear the fields before the operation is completed, place a redisplay statement immediately after the clear field statement.

clear field all

This statement clears all the displayed fields on the form. When the form is in Fill mode, it both clears displayed fields and assigns default values (although the default values are not displayed).

clear screen

This statement blanks out the entire terminal screen without clearing values from field data windows *only for the duration of the operation in which it is embedded.* It then redraws the screen and redisplays the data.

To control the number of seconds for which the screen is cleared, follow the clear screen statement with a sleep statement.

If the operation ends leaving the current terminal screen on display, the frame is redrawn. Typical uses of clear screen are to blank the frame temporarily while an operation is performed or before displaying a pop-up message, or to clean up as the application exits.

Examples

Clear the Name and Age fields in the current form:

```
clear field name, age;
```

Clear the field you specified in the field named Clearfield:

```
clear field :clearfield;
```

Clear all fields in the current form:

```
clear field all;
```

Present user with a blank screen for five seconds:

```
clear screen;
sleep 5;
```

Clear the screen in preparation for displaying a user prompt:

```
'Menu5' =
begin
    clear screen; /* Blanks screen */
    prmpt = prompt 'Enter the correct code';
    /* Screen redrawn here */
end
```

Clearrow

Clears a row or columns within a row of a table field or a record of an array.

Syntax

```
clearrow tablefieldname [ [integerexpr] ]
  [(columnname {, columnname})]

clearrow arrayname [integerexpr]
  [(attributename {, attributename })]
```

tablefieldname

Specifies the name of the table field whose row is being cleared. This is a 4GL name (unless you include *integerexpr* and its enclosing brackets).

arrayname

Specifies the name of the array whose record is being cleared

integerexpr

Specifies an integer expression indicating the row number in the table field or array

columnname

Specifies the name of a column in the specified row. This is a 4GL name.

attributename

Specifies the name of an attribute in the specified array record. This is a 4GL name.

Description

The 4GL clearrow statement clears an entire row of a table field or an entire record of an array, or specified columns within that row or record. The clearrow statement:

- Leaves a blank table field row or array record (unlike deleterow, which removes the row and closes up the table field or array).
- Clears all columns (except derived columns) unless you specify a *columnname*.

For table fields, the clearrow statement clears the table field row on which the cursor is currently resting, unless you specify an integer expression. In this case, it clears the row with that number. If you specify one or more columns as well, clearrow clears only those columns. If you have not specified an integer expression, make sure that the cursor is on a table field before executing the clearrow statement.

Because arrays do not have a current row, you must include as the integerexpr the index number of the row you want to clear. If you specify one or more attributes, clearrow clears only those attributes.

Example

Clear the columns called Cost and Total from the third row in the *partstbl* table field:

```
clearrow partstbl[3] (cost, total);
```

CloseFile()

Closes a file.

Syntax

```
[returnfield = ] [callproc] closefile
  (handle = handle
  [, disposition = disposition] )
```

returnfield

Specifies the name of field to which the result is returned. Integer.

handle

Specifies the file identifier used by the 4GL file access functions. Handle is an integer.

disposition

Indicates whether the file is preserved or removed when it is closed. The value is 'delete' or 'keep', or a 4GL expression that evaluates to one of these values.

Description

The closefile() function closes a file that was previously opened with the openfile() function. You can delete the file when you close it by specifying the clause disposition = 'delete'. The default is to keep the file.

The procedure returns 0 if the function completes without error.

Example

Close the file with the handle "file_no", and check for errors:

```
Status = callproc closefile (handle = file_no,
  disposition = 'keep');
 if status != 0 then
  message 'Unable to close file ' + ifnull(filename,' ')
  with style=popup;
  resume;
endif;
```

CommandLineParameters( )

Retrieves application-specific parameters from the operating system command line.

Syntax

```
returnfield = [callproc] CommandLineParameters( )
```

returnfield

A varchar variable into which the result is returned. If the application-specific parameters are too large to fit into this variable, they are truncated on the right.

Description

The CommandLineParameters( ) procedure retrieves an application-specific parameter that you specify on the command line when starting an imaged application. The parameter is retrieved into a varchar value of the size necessary to hold the parameter.

You can call this procedure from any frames or procedures within the application. You can call the function as many times as needed within an application; it always returns the same value.

If you allow a user to specify multiple parameters, the parameters are passed as a single string value. This string begins after the -a command line flag (described in the ABF part of this guide).

The parameters are concatenated with a single space separating adjacent parameters. You must be aware of any peculiarities of your operating system in the way it parses character strings into separate parameters (for example, in handling embedded quotes or multiple contiguous spaces).

You can use the SQL string functions described in the *SQL Reference Guide* to parse this string into the individual variables used by your application.

Although the CommandLineParameters() procedure is intended primarily for use with imaged applications, you also can use it with an interpreted application. In such a case, the first call to the procedure causes you to be prompted for up to 74 bytes of application-specific parameters.

See Examples for 4GL code that you can use to parse multiple command line parameters.

Examples

The following 4GL statement retrieves your department number as entered on the command line when you start the application:

```
deptno = callproc CommandLineParameters()
```

For example, you might enter the following command to start an application called Deptapp:

```
deptapp -a 567
```

In this case, the variable *deptno* contains the string '567.' (The space between the -a and the value is required.)

String Parsing Example

The following 4GL code uses built-in SQL string functions to retrieve an employee's department, division and building as entered on the command line. The data is stored in a varchar buffer called "empdata." In the example, "s" is a varchar work variable and "I" is an integer work variable.

```
Empdata = callproc CommandLineParameters();
s = squeeze ( empdata );
I = locate ( s, ' ' ); dept = left ( s, I - 1 ); s = shift ( s, -I );
I = locate ( s, ' ' ); div = left ( s, I - 1 ); s = shift ( s, -I );
I = locate ( s, ' ' ); bldg = left ( s, I - 1 ); s = shift ( s, -I );
if s != ' ' then
     message 'extraneous words entered in empdata ' + s
        with style = popup;
endif;
```

If you enter the following command to start the Empapp application:

```
empapp -a widget manufacturing 123A
```

Then the following values are returned:

- The variable *dept* contains 'widget'
- The variable *div* contains 'manufacturing'
- The variable *bldg* contains '123A'

See the *SQL Reference Guide* for more information about the string functions used in the above example.

Connect

Connects the application to a database, and, optionally, to a specified distributed transaction.

Syntax

```
connect [to] dbname
  [as connection_name]
  [session session_number]
  [identified by username | user username]
  [options = flag {, flag}]
  [with highdxid = value,   lowdxid = value]
```

dbname

Specifies the database to which the session connects. *Dbname* can be a quoted or unquoted string literal or a host string variable. If the name includes any name extensions (such as a system or node name), string literals must be quoted.

session_number

Specifies a positive integer constant whose value must be unique among existing session numbers in the application.

connection_name

Specifies a string of up to 128 characters that identifies the session. If you omit the as *connection_name* clause and the session clause, the default connection name is the specified database name. *Connection_name* must be specified using a quoted string literal or a host variable.

username

Specifies the user identifier under which this session runs. *Username* can be specified using a quoted or unquoted string literal or string variable.

flag

Specifies runtime options for the connection. Valid flags are those accepted by the sql command; flags specific to the Terminal Monitor are not valid. For more information about these flags, see the *Command Reference Guide.* The maximum number of flags is 12.

If you specify the -R flag and the role ID has a password, use the following format:

```
'-Rroleid/password '
```

The flags can be specified using quoted or unquoted character string literals or string variables.

value

Specifies a value: highdxid specifies the high-order 4 bytes of a distributed transaction ID; lowdxid specifies the low-order 4 bytes of a distributed transaction ID. These options are used for two phase commit of distributed transactions; for details, see the *SQL Reference Guide*.

Description

The connect statement connects an application to a database. By default, an ABF or Vision application connects to the database from which the application was linked, unless this was overridden by the -database or -nodatabase flag. The connect statement must precede all statements that access the database. To terminate a connection, use the disconnect statement.

Connecting with Distributed Transactions

The with clause identifies a specific distributed transaction by its distributed transaction ID. When you include the with clause, the application is connected to the specified database and the local transaction associated with the specified distributed transaction. This option enables a coordinator application to re-establish a connection with an open transaction if the connection between the application and the local DBMS server is lost for any reason.

The distributed transaction is identified by its distributed transaction ID, an 8-byte integer generated by the application. In the with clause, the *value* associated with highdxid must be the high-order 4 bytes of this ID and the *value* associated with lowdxid must be the low-order 4 bytes of the distributed transaction ID. The distributed transaction ID must have been previously specified in a prepare to commit statement.

When the program issues a connect statement that includes the with clause, either a commit or a rollback statement must immediately follow the connect statement. Commit commits the open local transaction; rollback aborts it. After the commit or rollback has been issued, the session can continue, issuing any valid statement. For more information about distributed transactions, see the *SQL Reference Guide*.

Creating Multiple Sessions

The connect statement establishes a *session*. If your application requires more than one session, you can assign a session identifier or number to each session, and use the set connection or set_sql(session) statements to switch sessions. For a detailed discussion of multiple sessions, see Multiple Session Connections (see page 910).

Using Session Identifiers

To assign a numeric session identifier to a connection, specify the **session** clause; for example:

```
connect accounting session 99;
```

assigns the numeric session identifier 99 to the connection to the "accounting" database. To determine the session identifier for the current session, use the inquire_sql(session) statement. To switch sessions using the numeric session identifier, use the set_sql(session) statement; for example:

```
set_sql(session = 99);
```

Using Connection Names

To assign a name to a connection, specify the as clause; for example:

```
connect act107b as accounting;
```

assigns the name "accounting" to the connection to the "act107b" database. To switch sessions using the connection name, use the set connection statement; for example:

```
set connection accounting;
```

If you omit the as clause, Ingres assigns a default connection name—the database specified in the connect statement. You can subsequently use this connection name in set connection statements to switch session. If you omit the as clause and specify a numeric session identifier (using the session clause), the default connection name is "ii*n*," where *n* is the specified numeric session identifier.

To determine the connection name for the current session, use the inquire_sql(connection_name) statement.

Permissions

This statement is available to any user; to use the identified by clause, you must be one of the following:

- The DBA of the specified database
- A user with the security privilege
- A user that has been granted the db_admin privilege for the database

Locking

The connect statement takes a database lock on the specified database. Unless you explicitly request an exclusive lock using the **-l** flag, the database lock is a shared lock.

Delete

Deletes rows from a database table.

Syntax

```
[repeated] delete from [owner.]tablename [corrname]
  [where qual]
```

[*owner*.]*tablename*

Specifies the name of the table from which the row is deleted, and the owner of the table. The table name can be a 4GL name; however, if a 4GL variable is used, an owner cannot be specified, unless the owner name is included as part of the value. In other words, the variable can contain the value *owner.tablename.*

corrname

Specifies the correlation name of the table

qual

Specifies a logical expression indicating which rows to delete from the table. It cannot include 4GL names. The **qualification** function is allowed.

Description

The delete statement removes rows from a table based on the qualification in the where clause, if present. Without the qualification, the statement deletes all the rows.

- You can use the qualification function in the where clause.
- You can store the where clause in a variable; however, in this case, you cannot use the qualification function.
- If you use the qualification function, the keyword repeated, if specified, is ignored.

Place a dereferencing colon [:] before the names of fields in the where clause so that the values in the fields are used and are not taken to be column names.

If you want to use the same delete statement frequently, consider using the repeated option for increased efficiency.

Deleting multiple rows from a table based on values from a table field normally involves the use of the unloadtable statement. For more information, see unloadtable Statement—Loop Through Rows and Execute Statements (see page 748).

Examples

Delete all rows in the Personnel table containing the employee number (*empno*) displayed in the current form, using the repeated option, then commit the changes:

```
repeated delete from personnel
  where personnel.empno = :empno;
commit;
```

Delete all rows in the Personnel table:

```
delete from personnel;
```

Delete all rows in the table specified in the *tablename* field:

```
delete from :tablename;
```

Delete rows from a table, where the table name and the where clause are expressed as variables:

```
whereclause = 'empno = 12';
tablename = 'personnel';
delete from :tablename
where :whereclause;
commit;
```

Delete rows from a table, where the table name is expressed as a variable and the where clause uses the qualification function:

```
delete from :tablename
  where qualification(empno =:field1,
    empname =:field2);
```

Deleterow

Deletes a row from a table field on the current form or a record from an array.

Syntax

```
deleterow tablefieldname [ [integerexpr] ]

deleterow arrayname [integerexpr]
```

tablefieldname

Specifies the name of the table field from which you are deleting a row. This is a 4GL name (unless you include *integerexpr* and its enclosing brackets).

arrayname

Specifies the name of the array from which you are deleting a record.

integerexpr

Specifies an integer expression indicating the number of the row to be deleted.

For a table field, the integer expression refers to a row in the table-field display rather than to a row in the data set. For an array, the integer expression refers to a record in an array.

As is usual for a 4GL name, *tablefieldname* derives its value from a field, a table-field column, or other variable not containing a row reference. If *tablefieldname* is derived from a table-field column, the value must be in the row on which the cursor is currently resting, not in a row location specified with a row number.

Description

The 4GL deleterow statement deletes a table-field row or array record.

For table fields, deleterow deletes the row on which the cursor is currently resting, unless you add an integer (*integerexpr*). In this case, it deletes the row with that number. For table fields, deleterow is not successful unless the cursor is on the table field or an integer expression is specified. For arrays, an integer expression must be specified.

The deleterow statement removes the specified table field row from the display. If the specified row had a state of Undefined or New, then deleterow also removes it from the table field dataset. If the specified row had a state of Unchanged or Changed, deleterow leaves it in the table field dataset, but changes its state to Deleted.

Deleterow does not validate the table field row before deleting it. To do this, precede the deleterow statement with a validrow statement.

For arrays, deleterow moves the array record to the beginning of the array, sets its state to Deleted, and changes its index to 0 (for the first deleted row) or the next highest negative integer. You can reference such records in either of the following ways:

- Explicitly by means of their new non-positive index
- Implicitly with the unloadtable statement

  The unloadtable statement processes deleted records after non-deleted records, even though the deleted records have smaller indexes. For more information on the order in which the unloadtable statement processes rows, see Processing an Entire Table Field or Array with Unloadtable (see page 870) and unloadtable Statement—Loop Through Rows and Execute Statements (see page 748).

Examples

Delete the current row in the table field:

```
deleterow partstbl;
```

Delete the third row in the table field:

```
deleterow partstbl[3];
```

Delete the seventh record of an array:

```
deleterow emparray[7];
```

Disconnect

Terminates access to the database.

Syntax

```
disconnect [current |connection_name |
    session session_identifier | all]
```

Description

The disconnect statement terminates a session connected to a database with the connect statement. It does not disconnect the default connection specified with the -database flag. The disconnect statement commits any pending updates.

To disconnect the current session, issue the disconnect statement with no arguments, or the disconnect current statement; other sessions remain connected. To switch sessions, use the set_sql or set connection statement.

To disconnect a session other than the current session in a multi-session application, you must specify the session *session_identifier* clause or the connection name. (Connection names and session identifiers are specified using the connect statement.) To determine the numeric session identifier for the current session, use the inquire_sql(:*session_id* = session) statement. To determine the connection name for the current session, use the inquire_sql(connection_name) statement. If you specify an invalid session, Ingres issues an error and does not disconnect the session.

Use the disconnect all statement to disconnect all sessions started with the connect statement; that is, all sessions with positive session numbers. The disconnect all statement does not disconnect any Ingres-generated sessions, including the initial or default session. Ingres-generated sessions are disconnected when the application exits.

Permissions

This statement is available to any user.

Locking

When the disconnect is issued, all locks held are dropped.

Examples

Disconnect from the current database:

```
Disconnect;
```

Disconnect a session in a multi-session application by specifying the connection name"

```
Disconnect accounting;
```

Disconnect a session by specifying its session identifier.

```
Disconnect session 99;
```

Display Submenu

Displays a submenu that is not attached to a query.

Syntax

```
display submenu
begin | {
  operations
end | }
```

Description

Display submenu displays on the menu line of a frame a menu that is not attached to a query. Use display submenu with an operation list if you want the form to be active; you have access to the fields on the form while the menu is displayed. The submenu is not influenced by form mode, and remains on the menu line until an endloop operation is executed.

The display submenu statement signals the start of a submenu that replaces the original menu line for the form. The submenu consists of the double keywords display submenu followed by an operations list (that is, a list of activations and statement blocks). The submenu operation can contain a time-out section and a database event section. The keyword end that concludes the display submenu statement must be followed by a semicolon.

As with submenus produced by attached queries, you can include an initialization block in a display submenu statement. This initialization operates the same way as in an initialization statement, except that you cannot declare variables in a display submenu initialization.

Display submenu allows you to display a submenu without attaching it to a query. If your 4GL specification does not require an attached query, do not construct one to use this submenu. 4GL supports two statements, run submenu and display submenu, that you can use in this way. The principle difference between the two statements is that, in display submenu, the form remains active, while in run submenu, the form is inactive. See Run Submenu (see page 1090).

You cannot use the resume menu or resume entry statements in a run submenu and its operations.

Example

This example displays a submenu with two menu options, DetailFrame and End:

```
display submenu
begin
  initialize =
  begin
    message 'Beginning submenu';
  end
  'DetailFrame' =
  begin
    callframe subframe;
  end
  'End' =
  begin
    endloop;
  end
end;
```

Endloop

Exits from a submenu and returns to the previous menu, or terminates a select, unloadtable or while loop, or a run submenu or display submenu statement.

Syntax

```
endloop [label]
```

label

Specifies an unquoted character string indicating which nested loop to end. It cannot be a field name or a keyword.

Description

The 4GL endloop statement can be used to break out of various sorts of loops. When used within a submenu created by an attached query or a run or display submenu statement, the endloop statement causes the application to exit from the submenu and return to the main menu of a frame. At the same time, endloop closes the attached query for the submenu.

Used within a select or while statement, endloop causes processing within the loop to terminate, transferring control to the first statement following endwhile or end. This is the only form of endloop that can optionally take a label as a parameter.

The *label* is useful within a nested series of while statements, to indicate which level of nesting to break out of. An endloop with a *label* causes control to break to the first statement following the endwhile for the labeled loop. See the while statement section for more information on labeled loops.

You can use the endloop statement to break out of an unloadtable loop without completing the processing of every row in the data set.

For additional information, see the sections in this chapter on the select (see page 1094), unloadtable (see page 748), while (see page 1169), run submenu (see page 1090), and display submenu (see page 1007) statements.

Examples

The End menu operation closes the submenu and its attached query:

```
'End' =
begin
  endloop;
end
```

Break out of the while loop labeled *B*:

```
endloop B;
```

Execute Immediate

Executes an SQL statement.

Syntax

```
execute immediate string_expression
```

string_expression

Specifies a character string that contains a valid SQL statement

Description

The 4GL execute immediate statement executes an SQL statement that you specify as the value of a string expression. You cannot specify a select statement, although the statement you specify can contain a subselect.

There are several other SQL statements that you cannot use with the execute immediate statement. See the *SQL Reference Guide* for more information.

Examples

Define a one-column view on the Employee table. The name of the view is stored in the varchar variable *viewname*, and the name of the column is stored in the varchar variable *columnname*.

```
Execute immediate 'create view ' + viewname
  + ' as select ' + columnname
  + ' from employee';
```

Execute Procedure

Invokes a database procedure.

Syntax

```
[returnfield = ] execute procedure [owner.]proc_name
   [(param_name=param_spec {, param_name=param_spec})]
```

return_field

Specifies an integer variable that contains the return value of the procedure

[*owner*.]*proc_name*

Specifies the name of the procedure being invoked. *proc_name* can be a literal name or a host string variable. You can specify an owner name when you use a literal value for the *proc_name*. When you use a host variable, the variable can contain either the procedure name or the *owner*.*proc_name* string. You cannot specify the owner and the procedure name in separate variables.

param_name

Specifies the name of the parameter being passed. *param_name* must be a valid Ingres name specified using a quoted or unquoted string or a host variable.

Each *param_name* must match one of the parameter names in the parameter list of the procedure's definition.

param_spec

The value of the parameter. The *param_spec* can be a literal value, a host variable containing the value to be passed (:*hostvar)*, or a host variable passed by reference (byref(**:***host_variable*)).

Description

The execute procedure statement executes the database procedure identified by *proc_name*. The execute procedure statement allows you to call a database procedure, even if the procedure has not been declared as part of an ABF or Vision application. You also can use a callproc statement to execute a procedure, but the procedure must be declared to ABF or Vision. You cannot specify an owner name when you use the callproc statement.

You can include execute procedure statements in a 4GL execute immediate statement, provided no parameters are passed using the byref clause. An execute procedure statement called by an execute immediate statement cannot return a status to the application.

Passing Parameters

You can pass parameters by value or by reference. By default, the execute procedure statement passes parameters to a database procedure by value. To pass a parameter by reference, use the byref() option.

By value

To pass a parameter by value, specify *param_name* = *value*. When you pass parameters by value, the database procedure receives a copy of the value. Values can be specified using:

- Numeric or string literals
- Ingres constants (such as today or user)
- Host variables
- Arithmetic expressions

The value assigned to a *param_name* must be compatible in type with the formal parameter represented by *param_name*. You can specify date data using quoted character string values, and money using character strings or numbers. If the data types are not compatible, Ingres issues an error and does not execute the procedure.

By reference

To pass a parameter by reference, specify the parameter as *param_name* = byref(:*host_variable*). When you pass parameters by reference, the database procedure can change the contents of the variable. Any changes made by the database procedure are visible to the calling program.

Exit

Closes the application.

Syntax

```
exit
```

Description

The 4GL exit statement closes the entire application and returns control to the level from which the application was originally entered.

Example

The Quit menu operation ends the application:

```
'Quit', key frskey2 =
begin
  exit;
end
```

Find_record()

Provides a search function for values in a table field data set.

Syntax

```
returnfield = [callproc] find_record(
  [form = formname,  ]
  [table_field =  tablefieldname,  ]
  [column =  columnname,  ]
  [value =  value, ]
  [noscroll = noscroll ] )
```

The following are optional keyword parameters for the find_record() procedure:

returnfield

Specifies the name of an integer field into which the result is returned

formname

Specifies a string expression whose value is the name of the form containing the table field to be searched. Default: the current form. If you specify *formname*, you must also specify both the *tablefieldname* and *columnname* parameters for find_record().

tablefieldname

Specifies a string expression whose value is the name of the table field to be searched. Default: the current table field. Required if the cursor is not on a table field. If you specify *tablefieldname*, you also must specify *columnname*.

columnname

Specifies a string expression whose value is the name of the column to be searched. This column must be of same type as *value.* Default: the column on which the cursor is positioned. Required if the cursor is not on a table field.

value

Specifies an expression (of any type) that specifies the value that is the object of the search. Must be of same type as column being searched. Can be a pattern-match string, if the column is a string type. If not specified, find_record prompts user for a value.

noscroll

Indicates whether to scroll to the matching data set record. If 0, scroll to the matching data set record. If non-zero, do not scroll. Can be used to find out if a record is in the data set without showing the record. The default is to scroll.

Description

The 4GL find_record() procedure searches a column of a table field for the first record after the current one that matches an input value, and scrolls to that row if the value exists. Find_record() can prompt you for this value or the programmer can specify it.

When find_record is passed a search parameter, the search is case sensitive and retrieves only an exact match for the value in a column or pattern-match if specified.

If no search parameter is specified, you are prompted for one. If your response:

- Contains a pattern-matching character, a case-sensitive search is performed
- Does not contain a pattern-matching character, a case-insensitive search is performed for a string that begins with your response

Find_record() optionally accepts keyword parameters to specify the form, table, table field, column, and an optional search value. If you do not specify a parameter, find_record() prompts for a search value and operates on the table field and column on which the cursor is currently resting.

Find_record() starts at the current row and wraps around to the beginning of the data set to continue the search when it reaches the end. If a value is found, find_record() returns the record number in the underlying table data set; otherwise it returns -1.

You can use pattern matching characters as input string values (as defined for the query language of the calling 4GL code). Use "\" as the escape character. For example, a 4GL/SQL frame can specify a pattern using SQL pattern matching characters.

If *value* is not specified, values you specify through a prompt behave slightly differently. The prompt value consisting of a single period (.) tells find_record() to use the last value entered through a prompt. This can be the search value for a different column and even for a different table field. To match a single "." Add "\" as an escape character. Unless you specify a pattern match character in the prompt string, the value is converted to a data type that matches the column to be searched. If the prompt string cannot be converted, the search is unsuccessful.

II_PATTERN_MATCH controls which pattern match characters are recognized. For details on setting II_PATTERN_MATCH, see the *System Administrator Guide.*

Finally, prompt strings used to search columns containing strings without pattern match characters perform an anchored match of the string column values. That is, the string value prefix matches the values in the column. This is equivalent to a pattern match of *string%* (or *string**).

Examples

Prompt the user for a value and search for it in the current column of the table field.

```
Callproc find_record();
```

Implement a Find function key:

```
key frskey7(explanation = 'Locate a
  specified string in the table field.') =
begin
  record_number = callproc find_record();
end
```

FlushFile()

Forces data to be written to disk.

Syntax

```
[returnfield =] [callproc] flushfile(handle = handle)
```

returnfield

Specifies the name of field to which the result is returned. Integer.

handle

Specifies the file identifier used by the 4GL file access functions. Integer.

Description

The flushfile() built-in function flushes the file to disk; that is, it writes the data from an open file to disk. The file must have been previously opened with the openfile() function and the file's handle must be known to 4GL.

You can flush a file of any type or mode. Do not flush a file in read mode.

Use the flushfile() statement to force a write to disk when the operating system cannot write to disk. Do not flush a file before closing it, because the closefile() statement automatically flushes the file.

The procedure returns 0 if the function completes without error.

Example

Flushes the file identified by the integer in the variable *fileno*.

```
Callproc flushfile(handle = fileno)
```

Help_Field()

Provides help on a field or column through the Field menu item of the help_forms statement.

Syntax

```
returnfield = [callproc] help_field()
```

Description

When the 4GL help_forms statement is executed, the Field menu item on the resulting frame calls the built-in procedure help_field(). Help_field() provides access to FRS help for the current field or column.

If a validation is available for the current field or column, help_field() provides a description of the validation. If the validation is of the form "field in [list of values]", help_field displays a table field containing the list of values, and allows you to select one. If no validation is specified, help_field describes the data type and field format for the current field or column.

Help_field does not accept any parameters. This procedure returns an integer indicating whether a help value was selected. This is a positive integer if a value was selected from a list. Otherwise, it is 0.

Example

Call the built-in procedure help_field().

```
Intvar = callproc help_field();
```

Helpfile and Help_Forms

Provides on-screen help information.

Syntax

```
helpfile subjectname filename

help_forms (subject = subjectname, file = filename)
```

subjectname

Specifies the name (subject) of the help text that Ingres displays. The name appears at the top of the window showing the help file. A 4GL name.

If the file name is not found, Ingres displays the message "Sorry- cannot open file on *subjectname*."

filename

Specifies the full specification for the file containing the help text for the current form. It can be any string expression.

Description

The 4GL helpfile and help_forms statements allow the designer to include a limited help facility in an application. Executing the helpfile or a help_forms statement displays the named file in the window in table-field format. The application user can scroll forward or backward to locate information using the cursor keys.

The helpfile statement restricts access to the information in the named file. It excludes access to help on field validations and function/control key mappings in the Ingres help facility.

The help_forms facility accesses the Help utility. It provides you with information on the current form, the fields of the form, or the current function/control key mappings. It also allows you to look up the validation criteria that are associated with any field.

The help_forms statement displays, as a menu option, the Field menu item. Field is generated by the built-in procedure help_field(). See Help_Field() (see page 1016).

Editing the Help Text File

As you develop a forms-based system that uses the help facility, it is useful to escape to a system text editor to edit the help text while running the application. The logical or environment variable II_HELP_EDIT provides this capability. By setting this name at the operating system level to any value (for example, true), it causes an extra menu operation, Edit, to appear in the help facility.

Choose the Edit operation to invoke the default system text editor on the current frame's help file. The following command, issued at operating system level, invokes the help-editing capability:

Windows:

```
set II_HELP_EDIT = TRUE
```

UNIX:

```
setenv II_HELP_EDIT TRUE
```

VMS:

```
define II_HELP_EDIT "TRUE"
```

See the ABF part of this guide for information about setting the default text editor.

Adding Subtopic Statements

You can specify a subtopic in a help file with the following format:

```
/# subtopic 'listitem' 'filename'
```

where *listitem* is the subtopic you want to be displayed and *filename* is the file containing the corresponding help text.

If you include one or more subtopic statements in a help file, Ingres displays a SubTopics menu item when you select Help. If you select SubTopics, Ingres displays a list of subtopics; the list consists of all the listitems from the subtopics statements in the help file. You can select a subtopic from the list: Ingres displays the help text from the corresponding file.

The Explanation Clause

When you use help_forms in a frame, the associated help/keys operation displays descriptive text on the menu items in a table field. The table field includes FRS and user commands, the keys to which these are mapped, and an explanation of each command. Menu items are always shown at the top of the table field. When the explanation clause is not used, the Explanation column of the table field is blank for frame menu items.

To enter the descriptive text in the help/keys table field, use the explanation clause in the menu activation. The syntax for the explanation clause is as follows:

```
'menuitemname' [ (explanation = string ) ]=
  { statement } ;
```

An example of the explanation clause follows:

```
'Add' (explanation = 'Add a new customer') =
begin
  insert …
end
 'Help', key frskey2 =
begin
  help_forms (subject = 'Maintain Customer',
    file = 'filename');
end
```

This example results in the following display in the **Help/Keys** table field:

Command	Control/Function Key	Explanation
Add	1	Add a new customer

Examples

The following examples show how to display a file from the Personnel frame.

Windows: Call the Ingres helpfile facility to display the specified file in the Personnel frame:

```
'Help' =
begin
   helpfile 'Personnel frame'
     '\usr\admin\files\personnel.txt';
end

Use the help_forms statement to display the specified file in the Personnel frame
 and to provide access to field validation criteria and function/control key mapp
ings:

'Help'=
begin
   help_forms(subject= 'Personnel frame',
    file =
    '\usr\admin\files\personnel.txt');
end
```

UNIX: Call the Ingres helpfile facility to display the specified file in the Personnel frame:

```
'Help' =
begin
  helpfile 'Personnel frame'
    '/usr/admin/files/personnel.txt';
end
```

Use the help_forms statement to display the specified file in the Personnel frame and to provide access to field validation criteria and function/control key mappings:

```
'Help'=
begin
  help_forms(subject= 'Personnel frame',
    file =
    '/usr/admin/files/personnel.txt');
end
```

VMS: Call the Ingres helpfile facility to display the specified file in the Personnel frame:

```
'Help'=
begin
  helpfile 'Personnel frame'
    'dra2:[usr.admin.files]personnel.txt';
end
```

Use the help_forms statement to display the specified file in the Personnel frame and to provide access to field validation criteria and function/control key mappings:

```
'Help'=
begin
  help_forms (subject = 'Personnel frame',
    file= 'dra2:[usr.admin.files]personnel.txt');
end
```

If-Then-Else

Chooses between alternative paths of execution.

Syntax

```
if condition then statement; {statement;}
  { elseif condition then statement; {statement;} }
  [else statement; {statement;}]
endif
```

condition

Specifies a Boolean expression with a value of TRUE, FALSE, or UNKNOWN (in the case of expressions involving Null values). This condition is tested to determine which set of statements is executed.

Description

The 4GL if-then-else statement combination establishes a variety of logical relationships, which you can use to control the flow of an application, through Boolean expressions. The if-then-else statement combination is the simplest form of flow-control statement, letting you choose between alternate paths of execution in response to the data you enter.

Boolean expressions can include comparison operators and the logical operators AND, OR, and NOT. Boolean expressions involving Null values can evaluate to Unknown. Any Boolean expression whose result is Unknown behaves exactly as if it evaluated to False.

You can nest two or more if statements. In such cases, each if statement must be closed with its own endif statement.

The following are If-Then-Else combinations:

if-then-endif

The if statement tests for the truth of an expression:

True

4GL executes the statements from the keyword then to the endif statement.

False or Unknown

4GL executes the statements after the endif statement.

if-then-else-endif

The if statement tests for the truth of an expression:

True

4GL executes the set of statements from the keyword then to the keyword else.

False or Unknown

4GL executes the statements from the keyword else to the keyword endif.

if-then-elseif-then-else-endif

The if statement tests for the truth of an expression:

True

4GL executes the set of statements from the keyword then to the keyword elseif.

False or Unknown

The interpreter tests another expression, which starts with the keyword else or elseif up to the keyword endif.

You can use one or multiple elseif statements.

Examples

Print message if no employee number was entered in the *empnum* field:

```
if empnum = 0 then
  message 'Please enter employee number';
  sleep 3;
endif;
```

Control the flow of frames within an application based on the value in the *status* field in the current form:

```
if status = n then
  callframe new;
elseif status = c then
  callframe completed;
else
  callframe inprogress;
endif;
```

Nest multiple **if** statements:

```
if status = 'n' then
  if empnum = 0 then
    message 'Please enter employee number';
    sleep 3;
  else
    callframe NewEmp
  endif;
endif;
```

Initialize

Sets the initial values in an ABF frame.

Syntax

```
initialize [([variable = typedeclaration
  {, variable = typedeclaration}]
  [tablefieldname.hiddencolumnname = typedeclaration
  {, tablefieldname.hiddencolumnname =
   typedeclaration}])] =
[ declare [ variable = typedeclaration
  {, variable = typedeclaration}]
  [tablefieldname.hiddencolumnname = typedeclaration
  {, tablefieldname.hiddencolumnname =
    typedeclaration}] ]
[begin | {
    statement; {statement;}
end | }]
```

The following are the parameters for the initialize and declare sections:

variable

Specifies the name of a local variable to hold temporary values in the form

tablefieldname.hiddencolumnname

Specifies the name of a hidden column within a table field on the form. Hidden columns must be of simple data type.

typedeclaration

Specifies the data type and length of the variable, record type name, or column. Use one of these formats:

```
simpletype [(length[,scale])] [with|not null];
recordtype;
array of recordtype;
array of type of table tablename;
array of type of form formname;
array of type of tablefield form.tablefield;
```

Description

The 4GL initialize statement declares any local variables in the frame and any hidden columns in the frame's table fields. This statement can also provide a set of 4GL statements to execute when the frame starts up.

The initialize statement is optional, but when it appears it must come first in the 4GL source file. It can include two sections, initialize and declare. The optional declare section follows the initialize section. Enclose start-up statements in the initialization section in a set of braces or the keywords begin and end.

Declarations name the local variables and hidden columns and specify their data types. See Data Types for Simple Local Variables and Hidden Columns (see page 817) for a description of data types in 4GL.

Variables and hidden table-field columns contain the data that the application manipulates, but are not displayed on the form. While they are invisible and inaccessible to you, they are accessible to the application in the same ways as are displayed fields and columns. A hidden column must be of a simple Ingres data type. A 4GL local variable can be of any Ingres- or user-defined data type.

4GL recognizes record type names in local variable declarations, which define records as instances of a record type and in references to these records and their attributes. All form fields and table field columns are accessible as keyword parameters for 4GL frames and procedures.

Initialize Section

Local variables and hidden columns declared in the initialize section can be accessed by calling frames, which can pass values into them.

Declare Section

4GL supports local variable declarations that are distinct from the parameter declarations in the initialize section. To declare variables whose scope is limited to the frame in which they are defined, use the keyword declare following the initialize section. Do not use the declare keyword unless you are declaring at least one local variable or hidden column.

You cannot pass values to local variables and hidden columns defined in the declare statement through the call parameter lists either positionally or by keyword; these variables and columns cannot be accessed by other frames. See Coding 4GL Procedures (see page 978) for more information.

Initialization of Fields and Variables

When you use a field name or column name as a variable in an expression, 4GL interprets the value at run time. When a form is first displayed, each field has a default value of either the empty string or zero, depending on the field's data type, with the following exceptions:

- If a default value was assigned during definition through the ABF FormEdit operation, the default appears when the form is displayed in Fill mode.
- If the value was passed into that field as a keyword parameter, that value appears.
- If the 4GL specification assigns a value to the field with the initialize statement, that value appears.

The field retains its value until changed by either a 4GL statement or by you. Initial values assigned to fields and variables are described in the following table:

Type of Variable	Nullable?	Field or Variable?	Starting Value
Any character type	Nullable	Field	Null
	Non-nullable	Field	Empty string
	Both	Variable	Empty string
Float	Nullable	Field	Null

Type of Variable	Nullable?	Field or Variable?	Starting Value
	Non-nullable	Field	0.0
	Both	Variable	0.0
Decimal	Nullable	Field	Null
	Non-nullable	Field	0.0
	Both	Variable	0.0
Integer	Nullable	Field	Null
	Non-nullable	Field	0
	Both	Variable	0
Date	Nullable	Field	Null
	Non-nullable	Field	Empty date
	Both	Variable	Empty date
Money	Nullable	Field	Null
	Non-nullable	Field	0.00
	Both	Variable	0.00
Record	N/A	Variable	Attributes initialized according to user-defined record type
Array	N/A	Variable	Empty

Example

Set up two local variables and three hidden table-field columns, using SQL data types, and give the simple fields initial values:

```
initialize (
  idnum = integer,
  idname = varchar(10),
  emptbl.empnum = integer,
  emptbl.empsal = float,
  projtbl.projid = varchar(6)) =
begin
  idnum = 0;
  idname = 'none';
end
```

Set a local array of type of table in preparation for using it in a procedure:

```
initialize =
declare
  parttable = array of type of table part;
  name = char(16);
  pos = smallint;
begin
  callproc selectparts
    (parttable = parttable);
end
```

Inittable

Changes the display mode and clears the data set of a table field.

Syntax

inittable ***tablefieldname*** ***modetype***

tablefieldname

Specifies the name of the table field being reinitialized. This is a 4GL name.

modetype

Specifies a 4GL name that specifies one of the following mode types available for displaying table fields:

read

Specifies that the user can view but not add or modify rows

update

Specifies that the user can modify but not add rows

fill

Specifies that the user can modify or add rows

query

Specifies that the user can add or modify row with query operators

Description

The 4GL inittable statement changes a table field's display mode and clears any associated data. When a frame first appears in an application, the form and any table fields are initialized to Fill mode, which permits the application user to enter values into them. Use the inittable statement to:

- Reinitialize a table field to a different mode
- Clear the named table field of all values

Inittable erases any values a table field receives from another frame through the parameter list of a callframe statement. Do not use inittable in the initialize block in this case. Instead, place the query that loads the table field in the current frame's initialize block, after inittable.

Both the inittable and mode statements can alter the effective mode of a table field. The mode statement changes the mode of the form as a whole. The mode of a table-field is set independently of the form as a whole, with this exception: When the form is in Read mode, the table fields in the form are also forced to behave as if they were in Read mode. This forced behavior only lasts until the form's mode changes out of Read mode. At this point, the individual modes of the table fields are in effect again. The inittable statement does not operate on hidden columns.

Examples

Clear the *emptbl* table field and change its mode to Update:

```
inittable emptbl update;
```

Clear the *emptbl* table field, changing its mode to the one specified in the field called *modetype* in the current form:

```
inittable emptbl :modetype;
```

Inquire_4gl

Returns the value of the behavior set with the set_4gl keyword clear_on_no_rows.

Syntax

```
inquire_4gl (field = clear_on_no_rows)
```

Description

This statement returns the value of clear_on_no_rows to the specified *field*. The *field* is a form field or variable of an integer data type.

The value of 0 (for off) means that the select target is left undisturbed if zero rows are returned. The value of 1 (for on) means that the select target is cleared if zero rows are returned.

Examples

Return the value of **clear_on_no_rows** to the variable *intvar*:

```
inquire_4gl (intvar = clear_on_no_rows);
```

InquireFile()

Inquires about a file.

Syntax

```
[returnfield =] [callproc] inquirefile(handle =
    handle
  [, filetype = byref(filetype) ]
  [, filemode = byref(filemode) ]
  [, filename = byref(filename) ]
  [, record = byref(recordno) |,
     offset =byref(offset) ] )
```

returnfield

Specifies the name of field to which the result is returned. Integer.

handle

Specifies the file identifier used by the 4GL file access functions. Integer.

filetype

Specifies the value can be text, binary, or stream. String.

filemode

Specifies a value: create, update, append or read. String.

filename

Specifies the name of the file. String.

recordno

Specifies the record number of the current location for a binary file. Integer.

offset

Specifies the offset of the current location. Integer.

Description

The inquirefile() built-in function returns the requested information about a file accessed by 4GL. The file must be open and the handle known to 4GL before information can be retrieved. The parameters must be passed by reference so the values can be returned.

The procedure returns 0 if the function completes without error.

Example

Inquires for information about the file identified by the handle specified in the variable *fileno*. Returns the location of the file pointer to the variable *offsetno*, and name of the file to the variable *filename*.

```
status = callproc inquirefile(handle = fileno,
    filename = byref(filename), offset=byref(offsetno))
```

Inquire_forms

Provides runtime information concerning the FRS.

Syntax

```
inquire_forms objecttype {parentname}
  (fieldname = inquire_forms_constant [(objectname)]
  {, fieldname = inquire_forms_constant
    [(objectname)]})
```

objecttype

Specifies the name of the form or field for which information is being obtained:

frs

Indicates FRS itself

form

Indicates a form

field

Indicates a simple field

row

Indicates a row in a table field

column

Indicates a column in a table field

menu

Indicates a menuitem on the form

table

Indicates a table field

The *objecttype* is a 4GL name and must be a character string.

parentname

Specifies the form in which a simple field or menu under inquiry resides or the form and table field in which a column or row under inquiry resides. This is a 4GL name and must be expressed as a character string.

A null string (' ') or a blank string indicates the current parent (that is, the form belonging to the current frame, or the table field on which the cursor currently rests). The *parentname* is not used when the objecttype for the statement is frs or form.

fieldname

Specifies the name of the field (with an appropriate data type) to hold the value of the *inquire_forms_constant.*

inquire_forms_constant

Specifies a keyword that identifies the information to be obtained from FRS. Also referred to as an FRS constant.

objectname

Specifies the name of the form or field for which a value is sought. This is a 4GL name and must be expressed as a character string. If omitted, the default is the current object. All forms or fields in the statement must have the same parent.

Description

The 4GL inquire_forms statement lets you retrieve information at run time from the FRS. The FRS manages many aspects of the forms that are displayed as an application runs. Typically, you use inquire_forms to retrieve information about your current form. You can only inquire on forms that have been displayed in the window during the current execution of the application.

If the application uses a submenu, you must place the inquire_forms statement prior to the submenu. If you inquire afterward, the value Undefined is returned. However, if your request applies to the submenu itself, place the inquire_forms statement within the submenu.

For forms, fields, and the FRS, Ingres provides a set of constants containing information that the application can use to perform conditional processing, position pop-up forms, handle errors, and so on. The inquire_forms statement returns a different set of constants for each form or table field for which information is requested. In addition to the general syntax shown above, the following sections cover the specific syntax and a list of constants for each type.

The set_forms statement sets the value of the constants, which you retrieve with inquire_forms. Most inquire_forms statements have corresponding set_forms statements. See Set_forms (see page 1133) for more information.

The discussion below is divided into two main sections, Inquire_forms *formobject* and Inquire_forms FRS, on the basis of *objecttype*.

- Inquire_forms *formobject* covers statements, which take one of these *objecttypes*: column, field, form, row, table or menu. The statements are presented in alphabetical order by *objecttype*.
- Inquire_forms FRS covers variations of the inquire_forms frs statement. The statements are presented in alphabetical order by *inquire_forms_constant*.

Inquire_forms Formobject

Inquire_forms *formobject* refers to the following variants of the inquire_forms statement:

```
inquire_forms column | field | form | row | table |
    menu
```

formobject can be one of the following: a displayed *column* within a table field, a *simple field*, a *form*, a *table*, or a displayed *row* within a table field. Each variant of the inquire_forms *formobject* statement is discussed in its own subsection below, in alphabetical order by form object type.

Inquire_forms Column

```
inquire_forms column formname tablefieldname
  (fieldname = inquire_forms_constant [(columnname)]
  {, fieldname = inquire_forms_constant
    [(columnname)]})
```

Use inquire_forms column to retrieve runtime information about a column. The constant here is the *objecttype* column within the table field in the current form. If formname and tablefieldname are null strings and columnname is not specified after the FRS constant (*inquire_forms_constant*), the constant refers to the column in the table field in which the cursor currently rests. The cursor must be positioned on a table field.

The following are legal values (the FRS constant or *inquire_forms_constant*) for inquire_forms column:

exists

Specifies an integer indicating whether the column exists or not. Returns 1 if the column exists; otherwise, returns 0.

name

Specifies a character string containing the name of the current column. Do not add a *columnname.*

number

Specifies an integer indicating the sequence number of the column within the table field (beginning with 1 at the far left)

length

Specifies an integer indicating the length of the data area in the column in bytes

inputmasking

Specifies an integer value indicating whether an input mask is in effect for the column. Returns 1 if inputmasking is on, 0 if inputmasking is off.

datatype

Specifies an integer value that represents the data type of the field.

integer=30
c=32
varchar=21
date=3
char=20
vchar=37
float=31
money=5
decimal=10

Nullable datatypes are returned as the negative value of their non-nullable counterparts. For example, a nullable integer datatype returns -30. Note that the datatype constant supersedes type.

derivation_string

Specifies a character string containing the derivation string for a derived column. Returns an empty string if the column is not derived.

derived

Specifies an integer value used to determine whether a column is derived. Returns 1 if the column is derived; otherwise, returns 0.

format

Specifies a character string containing the format string specified for the column.

valid

Specifies a character string containing the validity check for the designated column.

reverse, blink, underline, intensity, displayonly, invisible, normal

Integer values specifying whether the display attribute is on (1) or off (0)

color

Specifies an integer in the range from 0 to 7 indicating the color code for the column. The default is 0.

mode

Specifies a character string containing the column's display mode. Legal values: fill, query, or read.

Inquire_forms Field

```
inquire_forms field formname
   (fieldname = inquire_forms_constant [(fieldname)]
   {, fieldname = inquire_forms_constant
   [(fieldname)]})
```

This statement returns runtime information about a particular field. The *fieldname* following the *inquire_forms_constant* is the *frsobject* for this type of inquiry.

If *formname* is a null string and no *fieldname* appears after the FRS constant (*inquire_forms_constant*), the FRS constant refers to the current field on the current form.

Legal FRS constants for the *inquire_forms_constant* are described as follows. They apply to simple fields and table fields except as noted for invisible and change, which can be used with simple fields only.

exists

Specifies an integer indicating whether the field exists or not. Returns 1 if the field exists; otherwise, returns 0.

name

Specifies a character string containing the name of the current field. Do not add a *fieldname.*

number

Specifies an integer value containing the sequence number of the field within the form, specified during design of the form with VIFRED.

length

Specifies the length of the data window in bytes (an integer value). If the field is a table field, the value of length is 0.

inputmasking

Specifies an integer value indicating whether an input mask is in effect for the field. Returns 1 if inputmasking is on, 0 if inputmasking is off.

datatype

Specifies n integer value representing the data type of the field:

integer=30
float=31
c=32
vchar=37
date=3
money=5
char=20
varchar=21
decimal=10

Nullable datatypes are returned as the negative value of their non-nullable counterparts. For example, a nullable integer datatype returns -30. Note that the datatype constant supersedes type.

derivation_string

Specifies a character string containing the derivation string for a derived field. Returns an empty string if the column is not derived.

derived

Specifies an integer value used to determine whether a field is derived. Returns 1 if the field is derived; otherwise, it returns 0.

format

Specifies a character string containing the format string specified for the field. If the field is a table field, the value is an empty string.

valid

Specifies a character string containing the validity check specified for the field (empty for a table field)

table

Specifies an integer value indicating field type; set to 1 for a table field; otherwise, 0

reverse, blink, underline, intensity, displayonly, invisible, normal

Integer values specifying whether the display attribute is on (1) or off (0).

These are set using the ABF FormEdit operation to access VIFRED.

color

Specifies an integer in the range from 0 to 7, indicating the color code for the field. The default color code is 0.

mode

Specifies a character string containing the display mode of the field. The possible values are update, fill, query, or read for table fields and fill, query, or read for simple fields.

change

Specifies an integer value indicating whether you have made any changes to the field. Available only for simple fields. The value for change is 1 or 0 as follows:

1

If you have made any changes to the field, change is set to 1. The clearrest FRS command also sets change to 1. The clearrest command is mapped to the Return key on many terminals.

0

Change is set to 0:

- At the start of a display loop
- When the program places a value in the field
- When the field is cleared via a clear field statement.

Some FRS constants do not apply to a table field as a whole but rather to individual columns or rows. For these, use the inquire_forms column and inquire_forms row statements.

Inquire_forms Form

```
inquire_forms form
  (fieldname = inquire_forms_constant [(formname)]
  {, fieldname = inquire_forms_constant
  [(formname)]})
```

This statement returns runtime information about aparticular form. The following table shows legal values (the *inquire_forms_constant* or FRS constant) for the inquire_forms form statement:

exists

Specifies an integer indicating whether the form exists or not. Returns 1 if the form exists; otherwise, returns 0.

name

Specifies a character string containing the name of the current form. Do not add a *formname* as an operand.

change

Specifies an integer value set to 1 if changes have been made to the data displayed on the form. Otherwise, it is 0. A clear field all statement does not reset this to 0 for the form.

mode

Specifies a character string containing the display mode of the form. Legal values: none, fill, update, read, query.

field

Specifies a character string containing the name of the field on which the cursor is currently positioned.

rows

Specifies an integer value indicating the number of rows in the form specified by *formname*. If the form is a pop-up, the number indicates the rows occupied by the form's borders. The *formname* must be the name of the form where the cursor currently rests.

columns

Specifies an integer indicating the number of columns in the form named by *formname*. For pop-up forms, indicates the columns occupied by the form's borders. The formname must be the name of the form the cursor currently occupies.

Inquire_forms Menu

```
inquire_forms menu formname
   (fieldname = inquire_forms_constant[(menuitem)]
   {, fieldname = inquire_forms_constant[(menuitem)]})
```

Use this statement to inquire about the status of a menu item.

inquire_forms_constant

Specifies a constant:

active

Specifies an integer indicating whether the menu item is enabled (1) or disabled (0)

Inquire_forms Row

```
inquire_forms row formname tablename [rownumber]
  (fieldname = inquire_forms_constant [(columnname)]
  {, fieldname = inquire_forms_constant
    [(columnname)]})
```

To inquire about a particular displayed row, use an optional row number. The current row is the default. In an unloadtable statement loop, you can use inquiry only on the row just unloaded. If you specify a row number when using this statement with an unloadtable statement loop, you receive an error message.

If you do not specify a *formname*, *tablename*, *rownumber,* or *columnname*, then the constant refers to the current field within the table field in the form that the cursor currently occupies.

The following are legal values (the *inquire_forms_constant* or FRS constant) for inquire_forms row:

change

Specifies an integer value:

1

Indicates changes have been made to the data displayed in the column.

The clearrest FRS command also sets change to 1. The clearrest command is mapped to the Enter key on many PCs, and to the Return key on many terminals.

0

Indicates that change is set to 0:

- At the start of a display loop
- When the program places a value into the column
- When the program clears the column
- When a new row is created (as with an insertrow statement)

reverse, blink, underline, intensity

Integer values specifying whether the display attribute is on (1) or off (0).

These are set using the ABF FormEdit operation to access VIFRED.

color

Specifies an integer in the range from 0 to 7, indicating the color code for the field. The default color code is 0.

Inquire_forms Table

```
inquire_forms table formname
  (fieldname = inquire_forms_constant
    [(tablefieldname)]
  {,fieldname = inquire_forms_constant
    [(tablefieldname)]})
```

This variant retrieves runtime information about a table. If you leave formname as a blank and do not specify a tablefieldname after the FRS constant, the FRS constant refers to the table field of the form in which the cursor currently rests. Information returned about a table field always is at the table level rather than at the field level.

The following are legal values (the *inquire_forms_constant* or FRS constant) for inquire_forms table:

name

Specifies a character string containing the name of the current table field. Do not add a *tablefieldname*.

rowno

Specifies an integer value indicating the current row number within the table field, where the first row is numbered 1

maxrow

Specifies an integer value containing the number of displayed rows for the table field as defined with VIFRED

lastrow

Specifies an integer containing the number of displayed rows in the table field that actually contain data

datarows

Specifies an integer indicating the number of nondeleted rows stored in the data set for the table field

maxcol

Specifies an integer indicating the number of displayed columns defined for the table field

column

Specifies a character string with the name of the column, which the cursor occupies

mode

Specifies a character string containing the table field's initialization mode. Its possible values are read, update, fill, or query.

Inquire_forms FRS

The inquire_forms frs statement obtains information about the FRS according to the FRS constant in the statement syntax. The general syntax is:

```
inquire_forms frs (fieldname =
    inquire_forms_constant [(frsobject)]
    {, fieldname = inquire_forms_constant
      [(frsobject)]})
```

The following FRS constants can be used to produce different variants of the inquire_forms frs statement:

activate	errortext	outofdatamessage
columns	getmessages	rows
command	label	shell
cursorcolumn	last_frskey	terminal
cursorrow	map	timeout
editor	mapfile	validate
errorno	menumap	

The following subsections cover each variation individually.

Inquire_forms FRS (Activate)

```
inquire_forms frs (fieldname =
    activate [(activationtype)]{, fieldname =
    activate [(activationtype)]})
```

The inquire_forms frs (activate) statement checks whether activation is set to occur under the circumstances indicated by the activationtype parameter.

The *value* for each activationtype listed below is the integer 0 or 1, depending on whether the specified activation is turned off (0) or on (1).

Field activation only takes place if a field activation operation is specified for the current field.

The activate constant must be followed by one of the FRS objects (the *activationtype*):

before

Checks whether activation occurs upon entry into a field

keys

Checks whether activation occurs upon pressing a function or control key associated with a FRS key

menu

Checks whether activation occurs when you select the Menu key

menuitem

Checks whether activation occurs when you choose a menu operation

nextfield

Indicates that activation occurs when you move to the next field. The default is 1 (on).

previousfield

Indicates that activation occurs when you move to the previous field. The default is 0 (off).

Inquire_forms FRS (Columns)

```
inquire_forms frs (fieldname = columns)
```

The inquire_forms frs (columns) statement indicates the size, in number of columns, of the terminal screen. The columns constant does not accept a FRS object.

Inquire_forms FRS (Command)

```
inquire_forms frs (fieldname = command)
```

The inquire_forms frs (command) statement can be used in forms program activation blocks to determine what key stroke caused an activation to occur. The command constant does not accept a FRS object.

This statement can return one of the following values indicating the last keystroke made:

Value	Description
0	Undefined. Returned if an inquire statement is executed after exiting from a submenu block and before reentering the display loop.

Value	Description
1	Menu key
2	Any FRS key. To determine the number of the FRS key, see inquire_forms frs (last_frskey).
3	Any menuitem
4	Nextfield or autoduplicate
5	Previousfield
6	Downline
7	Upline
8	Newrow
9	Clearrest
10	Scrollup
11	Scrolldown
12	Nextitem
13	Timeout

Inquire_forms FRS (Cursorcolumn)

```
inquire_forms frs (fieldname = cursorcolumn)
```

The inquire_forms frs (cursorcolumn) statement indicates the current column position of the cursor in the window. The cursorcolumn constant does not accept a FRS object.

You can use the returned value to specify the starting column for positioning a pop-up message or lookup table. For example, you can use the value as input to the argument ii_startcolumn of the look_up( ) function.

Inquire_forms FRS (Cursorrow)

```
inquire_forms frs (fieldname = cursorrow)
```

The inquire_forms frs (cursorrow) statement indicates the current row position of the cursor on the terminal screen. The cursorrow constant does not accept a FRS object.

You can use the returned value to specify the starting row for positioning a pop-up message or lookup table. For example, you can use the value as input to the argument ii_startrow of the look_up( ) function.

Inquire_forms FRS (Editor)

```
inquire_forms frs (integer_var = editor)
```

The inquire_forms frs (editor) statement allows you to determine whether the system editor was disabled or enabled through a set_forms statement. A return value of 0 indicates that the editor is disabled, while a value of 1 indicates that it is enabled.

Inquire_forms FRS (Errorno)

```
inquire_forms frs (fieldname = errorno)
```

The inquire_forms frs (errorno) statement indicates the state of the error flag, which is set by FRS if an error occurs during the current display of the form. The errorno constant does not accept a FRS object. A return value of 0 indicates no errors. The error flag is set to 0 each time a menu operation is completed.

The errorno constant detects errors due to a user forms statement, such as getform or putform.

Inquire_forms FRS (Errortext)

```
inquire_forms frs (fieldname = errortext)
```

The inquire_forms frs (errortext) statement returns a string containing the text of the error message associated with the error number returned by *errorno*. If there is no current error it returns an empty string. The errortext constant does not accept a FRS object.

Inquire_forms FRS (Getmessages)

```
inquire_forms frs (fieldname = getmessages)
```

The inquire_forms frs (getmessages) statement allows you to determine whether the application has been set to suppress validation error messages. These are messages that occurs when the application attempts to retrieve a value from a field.

Inquire_forms FRS (Label)

```
inquire_forms frs (fieldname =
  label [(menuN|frskeyN|frscommand)]{, fieldname =
  label [(menuN|frskeyN|frscommand)]})
```

The inquire_forms frs (label) statement returns the alias for the control or function key (including arrow or shell keys) to which a menu item, FRS key or FRS command is mapped. The alias (a character string) is displayed in the Help facility and on the control/function key-mapped menu line. Default menu item labels are often set in the terminal mapping file.

The label constant accepts one of the following FRS objects:

menu*N*

Specifies the *n*th menu item on the menu line. *N* must be in the range 1 to 25.

frscommand

Specifies any of the FRS commands, including shell, to which a control or function key can be mapped.

frskey*N*

Specifies any of the FRS keys in use during the application. *N* must be in the range 1 to 40.

One of the *frscommand* options available for UNIX and VMS is shell.

UNIX: The *frscommand* spawns a Bourne shell (sh) (default) or c shell (csh) from within the forms program in a manner similar to starting an editor on a field's contents. For security it is not active unless the application explicitly enables it.

VMS: The *frscommand* spawns a DCL sub process from within the forms program in a manner similar to starting an editor on a field's contents. For security, it is not active unless the application explicitly enables it.

This statement can return the following FRS default values:

- If the specified FRS object is not mapped to any control, function, or arrow key, an empty string is returned.
- Where X is any one of the characters A through Z or the Esc or Del keys, control keys are returned as "ControlX."
- Where N is a number from 1 to 40, function keys are returned as "PFN." (See also inquire_forms frs (map).)

Inquire_forms FRS (Last_Frskey)

```
inquire_forms frs (integer_fld = last_frskey)
```

The inquire_forms frs (last_frskey) statement returns the number of the FRS key that caused an activation. When several FRS keys are combined into one activation block (to reduce duplicate code), use this syntax to determine, at run time, which specific key caused the activation block to be executed.

It is a good idea to execute the inquire_forms frs (last_frskey) statement as early as possible in an activation block. Information is lost if you place it after any forms statement that creates an on-screen display, such as message, prompt, run submenu, and display submenu. To preserve information, use a nested display submenu that saves FRS state information.

Inquire_forms FRS (Map)

```
inquire_forms frs (fieldname =
    map [(menuN|frskeyN|frscommand)]{, fieldname =
    map [(menuN|frskeyN|frscommand)]})
```

The inquire_forms frs (map) statement returns the alias, a character string containing the name of the control or function key (including arrow and shell keys) to which a menu item, FRS key, or FRS command has been mapped.

The map constant accepts one of the following FRS objects:

menu*N*

Specifies the *n*th menu item on the menu line. *N* must be in the range 1 to 25.

frscommand

Specifies any of the FRS commands to which a control or function key can be mapped.

frskey*N*

Specifies any of the FRS keys in use during the application. *N* must be in the range 1 to 40.

One of the *frscommand* options available is shell. For characteristics of this command, see Inquire_forms FRS (Label).

If no control, function, or arrow key is associated with the above FRS objects, map returns a null string. See the appropriate section of this guide for a complete explanation of function key mapping. (See also Inquire_forms FRS (Label).)

Inquire_forms FRS (Mapfile)

```
inquire_forms frs (fieldname = mapfile)
```

The inquire_forms frs (mapfile) statement returns a character string containing the specification for the application's key mapping file. The mapfile constant does not accept a FRS object.

Inquire_forms FRS (Menumap)

```
inquire_forms frs (fieldname = menumap)
```

The inquire_forms frs (menumap) statement indicates whether the menu line is set to display the function key equivalents (or assigned labels) for menu operations (Run(PF3), Edit(PF4), and so on). A return value of 1 indicates that the labels are displayed; 0 indicates they are not displayed. The menumap constant does not accept a FRS object.

Inquire_forms FRS (Outofdatamessage)

```
inquire_forms frs (fieldname=outofdatamessage)
```

The inquire_forms frs (outofdatamessage) statement indicates the current behavior for the "out of data" message. By default, the "out of data" message is displayed when a user attempts to scroll beyond the boundaries of a table field data set. The value can be one of the following:

0

Indicates the "out of data" message is off

1

Indicates the default behavior; the "out of data" message is displayed

2

Indicates that the forms system rings the monitor bell once instead of displaying the "out of data" message

Inquire_forms FRS (Rows)

```
inquire_forms frs (fieldname = rows)
```

The inquire_forms frs (rows) statement indicates the size, in number of rows, of the terminal screen. The rows constant does not accept a FRS component.

Inquire_forms FRS (Shell)

```
inquire_forms frs (value = shell)
```

The inquire_forms frs (shell) statement allows you to determine whether the shell command key is enabled or disabled. The *value* is 1 if the shell command is enabled, 0 if it is disabled.

Inquire_forms FRS (Terminal)

```
inquire_forms frs (fieldname = terminal)
```

The inquire_forms frs (terminal) statement returns a character string specifying the type of terminal being used. This information comes from TERM_INGRES. The terminal constant does not accept a FRS object.

Inquire_forms FRS (Timeout)

```
inquire_forms frs (fieldname = timeout)
```

The inquire_forms frs (timeout) statement returns an integer value indicating the number of seconds in the timeout period. A value of zero indicates there is no timeout period set. The timeout constant does not accept a FRS object.

Inquire_forms FRS (Validate)

```
inquire_forms frs (fieldname =
    validate [(validationname)]{, fieldname =
    validate [(validationname)]})
```

The inquire_forms frs (validate) statement indicates whether or not validation is set to occur for the specified FRS component. It returns an integer value equal to 1 if validation is triggered by the described action for that FRS component, or 0 if validation is not set to occur as a result of the specified action. While in effect, settings apply to all form fields in the application that have validation checks defined for them.

The following are possible values for *validationname*:

keys

Indicates whether fields are validated when you press a key mapped to a FRS key. The default is 0 (off).

menu

Indicates whether fields are validated when you press the Menu key. The default is 0 (off).

menuitem

Indicates whether fields are validated when you choose a menu operation. The default is 0 (off).

nextfield

Indicates whether fields are validated when you move to the next field. The default is 1 (on).

previousfield

Indicates whether fields are validated when you move to the previous field. The default is 0 (off).

Examples

Check whether validation occurs on moving the cursor backward through the form's fields:

```
inquire_forms frs
  (vprev = validate(previousfield));
```

Place the mode type of the current form into the field called *formmode*:

```
inquire_forms form (formmode = mode);
```

Get the sequence number in the current form of the *empnum* field and place its value in the *fieldno* field. Put the length in bytes of the data area for the *empname* field into the *fldlen* field:

```
inquire_forms field '' (fieldno = number
   (empnum),fldlen = length (empname));
```

Find the number of the FRS key selected by a user:

```
inquire_forms frs ( integer_fld =
  last_frskey );
```

Put the number of rows displayed on the current table field into the integer field called *numrows*:

```
inquire_forms table '' (numrows = maxrow);
```

Put the name and length of the data area of the current column on the current table field in the current form into the character field *colname* and the integer field *collength*, respectively:

```
inquire_forms column '' ''
  (colname = name, collength = length);
```

Check whether a change was made to the *rank* column of the current row:

```
inquire_forms row (changed = change(rank));
```

Get the terminal size in number of rows and columns:

```
inquire_forms frs (termrows = rows,
  termcols = columns);
```

Determine which FRS key caused an activation:

```
key frskey1, key frskey2 =
begin
  inquire_forms frs (fieldnum = last_frskey);
  if fieldnum = 1 then
    message 'Key 1 was pressed';
  else
    message 'Key 2 was pressed';
  endif;
 end
```

Inquire_sql

Provides diagnostic information during run time about various aspects of an application's interaction with the database.

Syntax

```
inquire_sql (fieldname = inquire_sql_constant
  {, fieldname = inquire_sql_constant})
```

You can use the inquire_sql statement to retrieve various types of information about a database event that has been raised. The following are the various inquire_sql constants related to events:

fieldname

Specifies the name of a field to hold the value of the Ingres constant

inquire_sql_constant

Specifies the constant containing the information about the results of a database statement

Possible values for *inquire_sql_constant* are:

connection_name

Specifies the name assigned to the session with the connect *dbname* as *connection_name* syntax. If the connection name was not specified and a session id was specified, Ingres assigns the session the connection name of ii followed by the session id. If neither the session id nor the connection name was specified in the connect statement, the connection_name is the database name. The connection_name is a varchar of length 128.

connection_target

Is the full specification of the database to which the application is currently connected. The specification is returned as the node name followed by two colons and the database name. For example, mynode::mydb.

dbmserror

Specifies the error number (positive integer) of the last query statement

endquery

Returns an integer value:

The last select loop statement returned a valid row.

1 = Indicates that the previous select loop statement was issued after the last row of the selected data. The data is invalid because no more information remained to be retrieved.

When endquery returns 1, the variables assigned values from the query are not changed.

errorno

Specifies the error number (positive integer) for any error that occurred on the last query statement. Cleared before each DBMS statement, so the value of errorno is valid only when the inquire_sql statement is issued immediately after the query statement.

0 = Indicates that no error occurred

errortext

Specifies a character string containing the error text of the last query. Valid only when the inquire_sql statement immediately follows the query statement. A character string result variable of size 256 can retrieve most Ingres error messages.

The returned error text is the complete error message of the last error, including the error number and a trailing end-of-line character. If the result variable is shorter than the error message, the message is truncated.

If there is no error message, a blank message is returned.

errortype

Returns genericerror (character string) if Ingres is returning generic error numbers to errorno. Returns dbmserror if Ingres is returning local DBMS error numbers to errorno. See your query language reference guide for information about the interaction of generic and local DBMS errors.

dbeventname

Specifies the name of a database event that was raised. The receiving variable must be a character string.

dbeventowner

Specifies the Ingres username of the user that created the database event that was raised. The receiving variable must be a character string.

dbeventdatabase

Specifies the name of the database in which the event was raised; this is always the current database. The receiving variable must be a character string.

dbeventtime

Specifies the date and time at which the database event was raised. The receiving variable must have an Ingres date format.

dbeventtext

Specifies the message text associated with the database event. The receiving variable must be a string of up to 256 characters. It must be of sufficient length to contain the message text; otherwise, the text is truncated.

messagenumber

Returns integer number of the last message statement executed inside a database procedure. The message text is determined by the user. If no message, a zero is returned.

messagetext

Indicates the message text of the last message statement executed inside a database procedure (character type). The message text is determined by the user. If the result variable is shorter than the message text, the message is truncated.

If no text, a blank is returned.

programquit

Returns integer value:

1 = If the FRS exits the application after any of the following errors:

- Errors resulting from attempting to execute a query without connection to a database
- Errors resulting from failure of the DBMS server while an application is running
- Errors resulting from failure of the communications server while an application is running

0 = The does not exit the application after the errors described above.

rowcount

Specifies the number of rows affected by the most recent query statement (an integer). Query statements: insert, delete, update, select. If the query is successful, the value of rowcount is the number of rows inserted, deleted, updated, or selected. If there are errors, or if statements other than these are run, the value of rowcount is negative. The following rules apply:

A *singleton query* always causes inquire_sql to return a value of 0, -1, or 1, because this query always terminates after finding the first matching row.

For *attached queries,* the value is the number of rows you actually view, controlled by the next statement in the submenu. The count() aggregate determines the number of rows qualifying for a query (for details, see your query language reference guide).

During execution of a *select loop* or *retrieve loop,* rowcount is undefined (except after the execution of a query language statement). After completion of the loop, the value of rowcount equals the number of rows through which the program looped.

session

Specifies the integer value assigned to the session with the connect session *session_id* statement. If the session parameter is not specified, Ingres assigns a unique session id to the session. The default connection is assigned to -1. Additional sessions are assigned positive integer values.

transaction

Returns integer value:

1 = A transaction is open

0 = No transaction is open

This information cannot be current.

Description

The 4GL inquire_sql statement provides various types of information about the state of the application at run time, including:

- The number of rows affected by the *last* query statement
- The error number returned by the *last* query statement, and the associated error text
- Status information about the current session and open transactions
- The number and/or text of a message statement executed inside a database procedure
- Information about a database event that has been raised

Inquire_sql returns the error number and rowcount from the database statement that precedes it. For this reason, be careful not to insert another database statement (such as a commit) between this query statement and the inquire_sql statement itself. For example, the order of statements must be insert (or other database statement), inquire_sql, commit.

By default, inquire_sql returns generic rather than local error numbers. This can be overridden with the logical II_EMBED_SET. You can also use the dbmserror parameter to inquire_sql. Variables declared to receive generic error numbers must be of integer type; smallint type is sufficient for local error numbers.

Example

Assign information about a deletion to the global variables called *rcount*, *errno*, and *txt*:

```
delete from employee
  where empnum = :empnum;

inquire_sql (rcount = rowcount, errno =
  errorno, txt = errortext);
if errno != 0 then
  message 'Error occurred on delete:' + txt;
  sleep 3;
elseif rcount =0 then
  message
   'No records with that employee number';
  sleep 3;
endif;
```

Check for the error number and display an error message:

```
insert into orders (order_no, cust_no, order_date, total, status)
 values (order_no, cust_no, date('today'), null, status) ;
 inquire_sql (errorno = errorno) ;
 if errorno != 0 then
  rollback ;
  message 'An error occurred saving changes.'
    with style = popup ;
  resume ;
 endif ;
 commit ;
/* assertion: no errors above;   */
/* commit changes */
```

Insert

Inserts rows into a database table.

Syntax

```
[repeated] insert into [owner.]tablename [(columnname
  {, columnname})]  values (expression
    {, expression}) | subselect
```

[*owner.*]*tablename*

Specifies the name of the database table into which you are inserting rows or records, and the owner of the table. The table name can be a 4GL name; however, if a 4GL variable is used, an owner cannot be specified, unless the owner name is included as part of the value. In other words, the variable can contain the value *owner.tablename.*

columnname

Specifies the name of a column within the database table into which you are inserting values. A 4GL name.

expression

Specifies an expression representing a value to be inserted (as the corresponding *columnname*).

An expression used in the values clause must usually be a constant, a field, a table-field column, or a database expression. *expression* can also be built from a function or an arithmetic operation using constants or single-valued fields. In addition, it can be built from a global or local variable, a literal, a record or an array record attribute.

subselect

Specifies a select statement that serves as the source of values to be inserted. For complete information see the *SQL Reference Guide*.

Description

The 4GL insert statement lets you insert rows into a database table, using values from the current form. If you plan to use the same insert statement frequently, use the repeated option.

The values can originate from simple fields, table field columns, variables, record or array attributes. To insert multiple rows of values from a table field into a table, place the insert statement inside an unloadtable loop. See the section Unloadtable for further information.

The 4GL insert statement is similar to the Interactive SQL insert. It requires either a values clause or a subselect statement. You can omit the list of database columns if the inserted values come from a subselect and if the column names in the subselect match the column names in the table. Otherwise, you can use a values clause to supply values.

When inserting values from a table field, the current row is assumed. To specify a different row, include the row number in brackets immediately following the table-field name.

The reserved word all, attached to the name of the current form or a table field in the current form, can substitute for the list of field names in the values clause. You can use all this way if each of the simple fields in the form or each of the columns in the table field corresponds in name and data type to a column in the table. An asterisk (*) is substituted for the usual list of database table columns. It is not necessary for each database table column to be mapped to a form object. Local variables and hidden table-field columns are ignored in the mapping.

Examples

Insert values of *projname* and *enddate* into the Projects table and then commit the changes:

```
insert into projects (name, duedate)
  values (projname, enddate);
 commit;
```

Insert all values in simple fields of Deptform into the Personnel table:

```
insert into personnel (*)
  values (deptform.all);
```

Insert all values in the third row of *partstbl* into *part* and then commit the changes:

```
insert into part (*)
  values (partstbl[3].all);
 commit;
```

Insert values in Projform into the table named in the Tablename field (fields and columns correspond):

```
insert into :tablename (*)
  values (projform.all);
```

Insert a computed value into the Personnel table:

```
insert into personnel (name, sal)
  values (name, salary*1.1);
```

Insertrow

Inserts a new row into a table field or a new record into an array.

Syntax

```
insertrow tablefieldname [ [integerexpr] ]
  [(columnname = expression{, columnname =
    expression})]
  [with(display_attr(columnname)=attr_value
    {, display_attr(columnname)=attr_value})]

insertrow arrayname [integerexpr]
  [( attributename = expression{, attributename =
    expression})]
```

tablefieldname

Specifies the name of the table field that is receiving a new row or record. This is a 4GL name (unless you have included *integerexpr* and its enclosing brackets).

arrayname

Specifies the name of the array that is receiving a new row or record

integerexpr

Specifies a value indicating the row number after which the new row is to be inserted. It refers to a row in the table-field display rather than to a row in the data set, or to a record of the array.

columnname

Specifies the name of a column in the new row or record; a 4GL name. Can be a string constant or string variable.

You can assign values to table-field columns with insertrow. If you omit the assignment of expressions to columns, the new row appears empty.

You can specify the special FRS attribute _state as a column name. Do not specify the FRS attribute _record. _State and _record are discussed in Writing 4GL Statements (see page 849).

attributename

Specifies the name of an attribute in the new record; a 4GL name. Can be a string constant or string variable.

You can assign values to array attributes with insertrow. If you omit the assignment of expressions to attributes, the new row appears empty.

You can specify the special FRS attribute _state as an attribute name. Do not specify the FRS attribute _record. _State and _record are discussed in Writing 4GL Statements (see page 849).

display_attr

Specifies the display parameter to be assigned to the specified column. The display attribute can be reverse, blink, underline, intensity or color.

attr_value

Specifies the value for the display parameter. For color, the value is an integer from 0 to 7. For other display attributes, the value is 1 to turn the parameter on or 0 to turn the parameter off. For more details, see the section, Set_forms (see page 1133).

Description

The 4GL insertrow statement inserts a new row or array record following the row or record you specify by an integer expression. If the expression evaluates to 0, the new row or record is inserted at the top of the table field or array. For table fields, you can omit the integer expression, in which case the row is inserted just after the row on which the cursor rests.

Insertrow and Table Fields

When adding rows to a table field, you are dealing only with visible rows in the window. The on-screen position or number has no relationship to the position the row might have in the underlying data set of the table field.

Assuming that the table field's mode is Fill, Update, or Query, the user can enter or change values in the new row. The cursor must be on the table field and an integer expression must not be specified.

Insertrow and Arrays

To perform operations on the records of an array, use the insertrow statement or direct assignment to the records of the array.

When adding records to an array, the insertrow statement inserts a new record after the record whose index number you specify. This number refers only to the location of the record in the array, and has no relationship to the number the record might have if displayed in the window in a table field.

For example, to load an array with constant values, specify several insertrow statements:

```
insertrow array [0]
  (coll = 1.0, col2 = 'Record1');
insertrow array [1]
  (coll = 2.0, col2 = 'Record2');
insertrow array [2]
  (coll = 3.0, col2 = 'Record3');
```

Instead of using the insertrow statement, load the data directly into each attribute of each array record:

```
array[1].coll = 1.0; array[1].col2 = 'Record1';
array[2].col2 = 2.0; array[2].col2 = 'Record2';
array[3].col3 = 3.0; array[3].col2 = 'Record3';
```

You must load the rows in sequence; you cannot have any empty rows. In the preceding example (for both the insertrow statements and the direct assignment), attempting to load data into *array[5]* without first loading *array[4]* results in a runtime error.

Setting the Row State

4GL accepts the word _state as a valid column identifier of type *integer*. You can set the table field row (or array record) _state (internal) through the following insertrow syntax:

```
insertrow tablefieldname |arrayname [ [ integer ] ]
  [ ( _state  = expression ) ]
```

The _state attribute of the new record is Unchanged unless you specify a value for it in the insertrow statement.

You can also set _state through a query to the database or through the unloadtable statement.

Whenever you specify a value for _state, it must be in the range 0 through 3 (for table fields), or 1 through 3 (for arrays). _State and _record are discussed in Writing 4GL Statements (see page 849).

Examples

Insert an empty row after the current row:

```
insertrow emptbl;
```

Insert a row after the third row in the table field, with a value of "new" in the *status* column:

```
insertrow emptbl[3] (status = 'new') ;
```

Insert a new record in an array and set **_state** to "new:"

```
insertrow emparray[5] (_state = 1)
```

The following statement opens up a new row immediately following row 4 in the table field *partstbl*.

```
insertrow partstbl[4];
```

The following statement inserts constant values into the third record of the array *emparray*.

```
insertrow emparray[2]
  (col1 = 3, col2 = "the third record");
```

The following statement inserts a new row into the first (top) index of the array *pricearr*:

```
insertrow pricearr[0]
  (name = 'bucket', cost = 20.00);
```

Loadtable

Adds a row of data after the last row of a table field's dataset.

Syntax

```
loadtable tablefieldname
  (columnname = expression {, columnname =
    expression} )
  [with(display_attr(columnname)=attr_value
    {, display_attr(columnname)=attr_value})]
```

tablefieldname

Specifies the name of the table field into which the values are to be loaded

columnname

Specifies the name of a table field column into which values are to be loaded

expression

Specifies the value to be loaded into the specified column

display_attr

Specifies he display parameter to be assigned to the specified column. The display attribute can be reverse, blink, underline, intensity or color.

attr_value

Specifies the value for the display parameter. For color, the value is an integer from 0 to 7. For other display attributes, the value is 1 to turn the parameter on or 0 to turn the parameter off. For more details, see Set_forms (see page 1133).

Description

The loadtable statement loads values into a table field's data set. Each execution of loadtable appends one row to the end of the data set.

The data loaded is displayed until the table field's display window is full. Then newly loaded values continue to be added to the end of the data set. You can see these rows by scrolling to them.

The loadtable statement loads values directly into specified columns in the table field. The list of *columnnames* identifies the columns receiving values and the values they are to receive. You can specify displayed, invisible or hidden columns, but not derived columns. Use the with clause to specify display attributes for each column.

Any column not included in the list of columnnames receives a null if it is nullable or a default value (blank for character columns and 0 for numeric columns). When possible, values for derived columns are calculated (based on the specified columns and values) and loaded into the data set.

You cannot use the loadtable statement with arrays. The following example shows how to append a row to an array:

```
idx = callproc ArrayLastRow(arr) + 1;
arr[idx].name = 'Mike';
arr[idx].empno = '2345';
```

Using the _state Constant

If you use the _state constant for columnname, the *value* must evaluate to one of the following:

0—UNDEFINED

1—NEW

2—UNCHANGED

3—CHANGED

If you load a row and specify a state of UNDEFINED, you must not assign any values to any non-hidden columns for that row. You cannot load a row with a state of DELETED.

By default, if you do not assign an explicit row state to a loaded row, the new row has the state of UNCHANGED. The state changes to CHANGED when you or the application alter any of the row's values. Each column in the loaded row (except hidden columns) then has its change variable cleared (set to 0).

Example

Selects from the "emp" table the employee number ("eno") and employee name ("ename") of all employees whose employee numbers are less than 30. The selected rows are appended to the end of the table field "emptf" without overwriting the existing contents of the table field.

The example uses the local variables "eno_wk" and "ename_wk." These must be declared with the same data types as the corresponding columns in "emp" and "emptf."

```
select eno_wk = eno, ename_wk = ename
from emp
where eno <= 30
begin
  loadtable emptf (eno = eno_wk,
      ename = ename_wk);
end;
```

Look_up()

Fetches field values from the database or an array so that the application user can select from a pop-up list.

Syntax

```
ret_val = callframe look_up
  ( ii_query =  query; |ii_array = array,
  columnname = byref (varname),
     {columnname = byref (varname),}
  ii_field1 = column {, ii_fieldn = column}
  [,ii_qualify = ' ']
  [,ii_field_titlen = fieldtitle ]
  [,ii_titles = 1 ]
  [,ii_title = title]
  [,ii_rows = rows]
  [,ii_startcolumn = startcolumn]
  [,ii_startrow = startrow] )
```

ret_val

Specifies an integer value determined as follows:

- If you use ii_query, *ret_val* is the number of the row in the data set that was chosen. If no row was chosen, *ret_val* is a non-positive integer (0 or a negative number).
- If you use ii_array, *ret_val* is the array index of the row that was chosen. If no row was chosen, *ret_val* is a non-positive integer (0 or a negative number).

ii_query

Specifies the 4GL select query that fetches the information to be displayed in the pop-up table field and returned to the *varname* parameters. Used when the lookup comes from the database.

ii_array

Specifies the array containing the information to be displayed in the pop-up table field and returned to the *varname* parameters. Used when the lookup comes from an array.

columnname

Specifies the field in the look_up() pop-up from which you select the value; determined as follows:

- If you use ii_query, then *columnname* is the name of a column returned by the query (one *columnname* is required).
- If you use ii_array, then *columnname* is the name of an attribute in the array (*columnname* is optional).

The data from *columnname* is returned to the corresponding varname.

varname

Specifies the 4GL variable or simple field to which the value of *columnname* is returned.

ii_field*n*

Specifies a string expression whose value is the name of the *n*th table field column to be displayed in the lookup table field, determined as follows:

- If you use ii_query, ii_field is the name of a column returned from the query.
- If you use ii_array, ii_field is the name of an attribute in the array.

You must include at least one field.

ii_qualify

If you specify ii_qualify = '', you are prompted whether to qualify the values in the query before fetching them. You cannot specify any other value than ''.

You are not prompted if you omit the keyword ii_qualify.

You can use ii_qualify with ii_query only.

ii_field_title*n*

Specifies an optional string that is used as the title for the *n*th column of the table field. If not specified when ii_titles is specified, the column name is used.

ii_titles

If you specify ii_titles = 1, the columns in the lookup table field are titled.

If you omit the ii_titles parameter, and do not specify any ii_field_title*n* parameters, the columns are not titled.

ii_title

Specifies an optional string to be used as a title for the whole displayed table field

ii_rows

Specifies the number of rows to be displayed in the table field. Default: 4.

ii_startcolumn

Specifies the starting column for the displayed pop-up lookup form. If not specified, the pop-up is displayed floating.

ii_startrow

Specifies the starting row for the displayed pop-up lookup form. If not specified, the pop-up is displayed floating.

Description

The look_up frame is a built-in function you can use in ABF applications. It allows you to implement a user-specified lookup of form values from the database or an array with a single call. The look_up frame is not available in QUEL.

The command retrieves all rows selected by the query (or all rows in an array), displays the values found in a pop-up table field with columns as specified by the ii_field values, and provides a selection menu. When the application user selects a row of the table field, look_up returns the value from the selected source column of the table to the variables (or fields on the current form) that you specify in the byref() clauses.

Look_up returns a status value that is the number of the record in the array or data set for the look_up query that was selected by the end user. If no records are selected, or you cancel the query, look_up returns 0 or a negative number.

The following table lists the keyword parameters for the look_up frame. To select a value, you must include either of the following:

- ii_query, ii_field1, and at least one *columnname* parameter
- ii_array and ii_field1

If you use ii_query, and the ii_qualify parameter is passed an empty string, the look_up frame qualifies the input query. Look_up() displays a qualification form that lists the displayed columns to be fetched and allows you to enter qualification views. This is in addition to any qualifying conditions specified in the where clause of the query.

To qualify a query, the frame displays a qualification form that lists the displayed columns to be fetched and allows you to enter qualification values for these columns. Only displayed columns are qualified; hidden columns are not qualified and thus are completely unknown to the end user.

The qualification form runs in FRS *query* mode, which enables the values to be qualified by operators entered along with the field values. It appears in the window as a pop-up form if it is small enough; otherwise it appears in full-screen size.

The look_up() qualification frame has the following menu items:

Go

Executes the lookup frame with the qualified values

Cancel

Cancels the operation and return to the previous frame

Help

Displays help for the frame

After 4GL qualifies and runs the query, the rows appear in a table field with the columns specified by the ii_field*n* keywords. Additional attribute values fetched by the query but not specified as display columns are hidden columns in the table field. That is, the query must specify and fetch all desired columns while the ii_field*n* keywords specify those that are displayed. This table field appears in pop-up form if it is small enough.

By default, if you specify ii_titles = 1, the titles of the columns are the same as the names of the columns fetched from the database (if you specified ii_query) or the names of the attributes of the array (if you specified ii_array). Specifying the title columns by the values in the ii_field_title*n* keywords is optional. No column titles are displayed if neither ii_titles nor any optional column titles are specified.

If you specify ii_query and the query does not fetch any rows, the "No rows retrieved" message appears and the look_up frame returns without setting any of its input parameters. If only one row is fetched, the lookup function returns immediately without displaying the row and sets any *varname* parameters with values from that row. The look_up frame runs this form with the following menu items:

Select

Selects the current row

Cancel

Returns to the calling frame without returning any values

Help

Displays Help for this frame

The Select menu item returns the values of the current row by reference (including hidden columns) to the calling frame as specified by the named column parameters. These named parameters, which correspond to each of the columns fetched by the query, identify global variables in the look_up frame from which values can be returned when a row is selected.

You can use by reference expressions to manipulate the values in fields, table field columns, record attributes, and array record attributes by referencing the desired component.

Examples

The following example fetches all the rows of the database table Description, which has columns *name*, *id*, and *description*. It displays the values of the *name* and *description* columns in a pop-up table field along with a selection menu. When you select a row of the table, the statement returns the value from the *id* column of the table corresponding to the selected row into the field Proj_id on the current form. Note that commas are used as parameter list separators.

```
callframe look_up
  ( ii_query = select * from description;
    id = byref(proj_id),
    ii_field1 = 'name',
    ii_field2 = 'description');
```

For an example of filling a lookup table from an array, see the confirm procedure in the Employee sample application chapter, Sample 4GL Application (see page 1187).

Message

Prints a message in the window.

Syntax

```
message stringexpr
  [with style = menuline | popup [(option = value
  {,option = value})]]
```

stringexpr

Specifies a character string containing the message to be displayed. If the character string is a constant, enclose it in quotes. The string is of the varchar datatype, and can be a 4GL name.

option

Specifies startcolumn, startrow, columns, or rows; used in the with style clause to locate and create a pop-up

value

Specifies the value of each of the pop-up style option parameters listed above.

The values for startcolumn and startrow specify the coordinates of the upper left corner of the pop-up form. The value can be a keyword, integer constant, or integer variable as follows:

default | 0

Indicates the VIFRED definition or default

int

Specifies a positive integer indicating the number of rows or columns from the origin. The origin (1,1) is the upper left corner of the screen.

intvar

Specifies a 4GL expression evaluating to 0 or a positive integer

Description

The 4GL message statement causes a character string to appear in the window, either on the menu line or as a pop-up frame.

Use the with style option to specify where the message appears.

- To display the message on the menu line (at the bottom of the window), use with style = menuline or omit the with style clause.

  If you use with style = menuline, you can specify the length of time a message remains in the window. Place a sleep statement immediately after the with style = menuline statement, and the message remains visible for the number of seconds specified. See the sleep statement.

  If a query statement immediately follows a menu-line style message statement, the message remains in the window until the action is completed. Menu-line style messages are truncated, if necessary, to fit the runtime screen width.

- To display the message in a pop-up box at a window location you specify, use with style = popup. If there is not enough room in the window to start the pop-up box at the specified location, the forms system adjusts the location to show the entire pop-up box, if possible. If the pop-up box is taller or wider than the window, then the forms system displays it in full-screen mode.

  If you use with style = popup, the instruction "[Press Return]" is automatically added to the message. The message stays on display until you clear it by pressing the Return key.

  Pop-up style messages are truncated, if necessary, to fit the specified box size. If no box is specified, a box large enough to accommodate the message text is drawn.

  The *value* of rows, if specified, must be at least 4. Three of the rows are used by the display, leaving one line for text. If you want to display four lines of text, specify seven as the *value* of rows. The *value* of columns, if specified, must be at least 16.

Examples

Display a menu-line style message for three seconds:

```
message 'Please enter employee number';
sleep 3;
```

Display a message on the menu line until a database retrieval is completed:

```
message 'Retrieving employee name';
empform := select lname, fname from emp
  where empnum = :empnum;
```

Display a pop-up style message on the number of rows retrieved:

```
empform := select lname, fname
    from emp
    where empnum = :empnum;
inquire_sql (rcount = rowcount);
message 'Retrieved ' + varchar(:rcount) +
    ' rows' with style=popup;
```

Mode

Sets the current form to the designated display mode.

Syntax

```
mode 'modetype'
```

modetype

Specifies a 4GL name that specifies one of the following display mode types. The modetype must be enclosed in quotes.

fill

Clears all visible fields except for default values specified when the form is edited with VIFRED. In Fill mode, you can enter or change data. Default mode.

update

Allows you to enter or change data but does not clear fields before initialization

read

Allows you to look at but not change the data on the form. Certain letters, when typed in this mode, behaves as if typed in conjunction with the Control key. For example, typing p produces Ctrl-p. Exit activations do not occur if the form is in read mode.

query

Presents a blank form in which you can enter data to build a query on the database. This mode allows you to enter comparison operators (=, >, <=, and so on) as well as data. A form must be in query mode to allow the use of the qualification function in a database query. Derived fields are not active in Query mode.

Description

The 4GL mode statement typically appears in the initialization section of a frame to set the display mode for the form associated with that frame. The mode statement reinitializes the frame; it breaks the current display and redisplays the current form as if it were being entered from the top in the new mode.

The mode statement also affects the contents of simple fields or the way the table fields behave, depending on the mode you select. It does not reset the mode for table fields. These and other effects of the mode statement are discussed in detail below.

To change a form's mode without reinitializing the form, use the set_forms statement. The set_forms frs(mode) statement is generally better for setting the mode for a form. For more information, see Set_forms (see page 1133).

You can issue both the mode and set_forms statements from anywhere in the frame's 4GL code to change the mode during submenu execution. Upon exiting a submenu, the form reverts to the mode in effect prior to the submenu execution.

Note that if you change to query mode and must retain the values in some of the fields, you can assign the values to local variables, execute the mode query statement, and then assign the hidden values to the original fields and execute the redisplay statement.

Summary of Effects

The following chart summarizes the effects of the mode statement:

Possible Action	**Effect**
Clears simple fields:	
In Fill mode?	Reinitializes to VIFRED defaults
In Update mode?	Leaves fields unchanged
In Read mode?	Leaves fields unchanged
In Query mode?	Clears fields
Clears values in simple fields with attribute *Keep Previous Value* (set in VIFRED):	
In Fill mode?	No
In Update mode?	No

Possible Action	Effect
In Read mode?	No
In Query mode?	Yes
Clears table fields?	No
Clears global variables?	No
Reruns initialize block?	No
Resets the forms constant change to 0?	Yes
Ends an Unloadtable loop?	No
Ends an attached query (submenu)?	No. After the attached query, the form returns to the mode in effect prior to the submenu.
Allows entry and exit activations to occur:	
In Fill mode?	Yes
In Read mode?	No
In Query mode?	No

Examples

Put the current form in fill mode:

```
mode 'fill';
```

Change the display mode according to the value of the *modetype* field:

```
mode :modetype;
```

Save, in local variables, the values currently displayed in the *name* and *date* fields, change display mode to query, and redisplay the values saved in the local variables:

```
tname := name;
tdate := date;
mode 'query';
name := tname;
date := tdate;
redisplay;
```

Next

Displays the next row of retrieved data on the form.

Syntax

```
next
```

The next statement has no parameters.

Description

The 4GL next statement is useful when a database retrieval returns more than one row of values into a form that can hold only a single row at a time, as in an attached query with a submenu or query loop. By activating a menu operation that includes the next statement, a user can display the subsequent rows of data. When the retrieval occurs in the parameter list while calling another frame, the next statement normally appears as part of a menu operation in the called frame. If the retrieval occurs within a frame, the next statement appears in a submenu.

The next statement causes subsequent rows of data to be displayed until no more rows are found. After all rows have been displayed, executing the next statement leads to this message:

```
No more rows
```

The main menu, if it is also the current menu, is then redisplayed. If the current menu is a submenu, the main menu for the frame appears, and control passes to the statement immediately following the original assignment statement that contained the submenu. For more information on submenus, see Submenus (see page 1117).

In a query on a Master/Detail join, using the next statement causes the next master row to be displayed along with all its associated detail rows. For more information about Master/Detail queries, see Master/Detail Query (see page 1115).

Example

Select employee information one row at a time into the simple fields of Empform. This is an example of an attached query with a submenu.

```
empform := select lname, fname, empnum
  from employee
  where empnum <= 100
  begin
  /* submenu statements here including: */
    'NextRow' =
    begin
      next;
    end
  end;
```

OpenFile( )

Opens a file for writing.

Syntax

```
[returnfield =] [callproc] openfile(filename =
  filename
  [, filetype = type]
  [, filemode = mode]
  [, recordsize = size]
  [, delimiter = delim]
  , handle = byref(handle) | handle)
```

returnfield

Specifies the name of field to which the result is returned. Integer.

filename

VMS: Specifies the name of the file to be opened. The *filename* is a string expression containing the full path or a filename. *filename* is a string.The *filename* can be a logical name.

type

Specifies a type: text, binary, or stream. The default is text. *type* is a string. See File Types (see page 901) for a more detailed explanation of types.

mode

The value can be create, update, append or read. The default is read. *mode* is a string.

If the file does not exist, openfile( ) does not create a file unless the *mode* is create.

See File Modes (see page 902) for a more detailed explanation of modes.

size

Specifies the size, in bytes, of the records in the file. This parameter is optional and applies only to binary files. Integer.

If you do not specify *size*, 4GL calculates the record size based on the item list for the first read or write to the file.

The item list for each read or write must be the same size as the specified or calculated record size. If the total of the items is not the same as the record size, the read or write fails.

The record size is not stored when the file is closed and reopened.

delim

Specifies the delimiter used to separate data in a text file. It can be any printable character. The default is no delimiter.

handle

Specifies the file identifier used by the 4GL file access functions. Integer.

Description

The openfile() built-in function opens the specified file for reading or writing. The file must be on a local node.

The procedure returns 0 if the function completes without error.

PositionFile( )

Positions a binary or stream file to the specified offset or record, relative to the base.

Syntax

```
[returnfield = ] [callproc] positionfile(handle =
  handle, offset = offset | record = recordno
  [ , base = base] )
```

returnfield

Specifies the name of field to which the result is returned. Integer.

handle

Specifies the file identifier used by the 4GL file access functions. Integer.

offset

Specifies the location, in bytes, from the specified base. If the base is not specified, the location is measured from the beginning of the file.

For a binary file, the positionfile() function always positions the file to the beginning of a record. 4GL adjusts the file position to the beginning of the record containing the byte number specified by *offset*.

Can be a positive or negative integer, including zero. Integer.

recordno

Specifies the location, in record numbers, from the specified base. If the base is not specified, the location is measure from the beginning of the file. Can be a positive or negative, non-zero integer. Applies to binary files only. Integer.

base

Specifies the location to start measuring the offset or record from. The base can be 'start', 'end', or 'record'. The default is base = 'start'. String.

Description

The positionfile() built-in function positions a binary or stream file. You specify the position by *offset*, the location in bytes, or by *recordno*, the record number. You must indicate the position by specifying a record number or the amount of offset, but not both.

The offset or record number is relative to the position you specify by the keyword base. The base can be the current position, or the beginning or end of the file. If you do not specify the base, the record number or offset is measured from the beginning of the file.

The *recordno* indicates the number of records forward or backward from the base. A record is a logical division only, which is determined by the file's record size. You can only use the record keyword for binary files.

The *offset* indicates the number of bytes forward or backward from the base. For a binary file, the positionfile() function always positions the file to the beginning of a record. If you specify the offset clause, 4GL adjusts the file position to the beginning of the record containing the specified byte number.

You cannot specify a value for offset or record that moves beyond the beginning or end of the file.

You can only position a file that was previously opened with the openfile() statement. The file must be open and the handle known to 4GL. You cannot position a file that was opened in append or create mode.

You cannot position a text file, but you can rewind it with the rewindfile() statement.

The procedure returns 0 if the function completes without error.

Examples

In a binary file, records are logical divisions only; that is, there is no physical record delimiter between records. In the examples in this section, the file is a binary file, with a record size of 41 bytes. The beginning of each record is determined from the record size, as shown in the following table:

0	record 1	
41	record 2	
82	record 3	
123	record 4	
164	record 5	<-- CURRENT FILE POINTER
205	record 6	

246	record 7
287	record 8

The following examples position the file to the beginning of the file (byte 0):

```
positionfile(handle=handle, offset=1, base='start')
```

or

```
positionfile(handle=handle, record=1, base='start')
```

The following example also positions the file to byte 0, because the offset is adjusted to be the beginning of a record:

```
positionfile(handle=handle, offset=39, base='start')
```

In the following example, the file position does not change. The file pointer remains at byte 164:

```
positionfile(handle=handle, offset=20, base='current')
```

The following example positions the file to byte 123:

```
positionfile(handle=handle, offset=-1, base='current')
```

The following example positions the file to byte 164, because byte 187 is in the record beginning at byte 164.

```
positionfile(handle = handle, offset=-100, base ='end')
```

The following examples position the file to record 7 (byte 246):

```
positionfile(handle=handle, record=7, base='start')
```

or

```
positionfile(handle=handle, record=-1, base='end')
```

The following example positions the file to the 3rd record, offset 82:

```
positionfile(handle = handle, record=-5, base ='end')
```

Printscreen

Prints or stores in a file a copy of the current frame and its data.

Syntax

```
printscreen [(file = filename | 'printer')]
```

filename

Specifies the name of a file to the end of which a copy of the frame and its data are appended. It is a character string expression.

'printer'

Specifies a quoted string that causes the file to be sent directly to a printer utilizing the operating system print command

Description

The 4GL printscreen statement either prints or stores in a file a copy of the current frame and its associated data. To store the copy in a file, specify a valid file name. The copy is appended to the end of the current contents, if any, of that file. If you specify the quoted string 'printer', the copy goes directly to a printer.

If you omit the parameter to the command, the copy goes to a default file (defined with the logical/environment variable II_PRINTSCREEN_FILE) if one exists. Otherwise, Ingres prompts you for the name of the file. You enter the name of the file, or specifies "printer" to print the copy.

Examples

Send a copy of the current frame to the printer:

```
printscreen (file = 'printer');
```

Store a copy of the current frame in the file designated in the *printfile* field of the form:

```
printscreen (file = printfile);
```

Store a copy of the current frame in a file called *screen.txt*:

```
printscreen (file = 'screen.txt');
```

Prompt

Displays a character string in the window and accepts input from the application user.

Syntax

```
fieldname = prompt [noecho] charstringexpr
  [with style = menuline | popup [(option=value
  {, option=value})]]
```

fieldname

Specifies the name of a field (of character data type) to which your response is assigned. Can be a form field or local variable. If the response is longer than the field, the response is truncated on the right to fit.

noecho

Indicates that your response to the prompt must not be displayed in the window as it is typed. If *fieldname* is a displayed field, the response appears in the field.

charstringexpr

Specifies a string expression with a maximum length of 255 characters for the display of a message on the screen.

option

Specifies one of four options—startcolumn, startrow, columns, and rows—used in the with style clause to locate and create the pop-up box for displaying a prompt.

value

Specifies the *value* assigned to each popup style *option* parameter, specifying one of the following for the pop-up box: starting column position, starting row position, number of columns, or number of rows (including borders).

The *values* for startcolumn and startrow specify the coordinates of the upper left corner of the pop-up form. The *value* can be a keyword, integer constant, or integer variable as follows:

default | 0

Specifies the VIFRED definition or default

int

Specifies a positive integer indicating the number of rows or columns from the origin. The origin (1,1) is the upper left corner of the screen.

intvar

Specifies a 4GL expression evaluating to 0 or a positive integer

The *value* of rows, if specified, must be at least 4. Three of the rows are used by the display, leaving one line for text. If you want to display four lines of text, specify seven as the *value* of rows. The *value* of columns, if specified, must be at least 16.

Description

The 4GL prompt statement enables the application to request your input. It displays a string in the window, accepts your input, and places your response in a designated field.

- To display the prompt on the menu line, use with style=menuline or omit the style clause.
- To display the prompt in a pop-up box at a window location you specify, use with style=popup. If there is not enough room in the window to start the pop-up box at your specified location, the forms system displays the form in full-screen width. If you do not specify a location for the pop-up, it is displayed at the bottom of the frame.

Examples

Prompt for report name and place response in the local variable called *answer*:

```
answer := prompt 'Enter the report name: '
```

Prompt the user for a password:

```
password := prompt noecho
  'Enter your password: '
```

Prompt the user, using the pop-up style display, for the department of the person whose name appears in the Fname and Lname fields:

```
answer := prompt
  'Enter the department for ' + fname +
  ' ' + lname + ':'
  with style=popup;
```

ReadFile( )

Reads a file.

Syntax

```
[returnfield = ] [callproc] readfile(handle = handle
  , byref(item) {, byref(item) } )
```

returnfield

Specifies the name of field to which the result is returned. Integer.

handle

Specifies the file identifier used by the 4GL file access functions. Integer.

item

Specifies the data that you are retrieving from the file. String. Reads in as many bytes as are in the item datatype.

Description

The readfile() built-in function reads the data in a file that was previously opened with the openfile() function. The file must be open and the file's handle must be known to 4GL.

For a binary type file, the total size of the items read from a file must be the same as the file's record size. You can declare the record size in the openfile() statement, otherwise 4GL calculates it from the item list size indicated with the first read or write to the file.

4GL processes only string data types for files opened as a text file. 4GL issues an error if the item list returns data of types other than string.

The procedure returns 0 if the function completes without error.

Redisplay

Refreshes the frame and displays the current values.

Syntax

```
redisplay
```

Description

The 4GL redisplay statement refreshes the frame and displays the current values earlier than normally occur. Usually, changes made to fields don't appear in the window until all statements for the current operation are executed.

The forms system does not process a redisplay statement if no changes have been made to the current frame, or if the forms system is unable to detect those changes (such as, changes made to the frame via write statements in an Embedded Query Language procedure).

To force the forms system to refresh the frame, issue a clear screen statement prior to the redisplay statement.

There are no parameters for redisplay.

Example

Display a newly created part number before updating database table *part* with the values displayed in the form:

```
'Add' =
begin
  editparts := select partno =
    (max(partno) + 1)
    from part;
  commit;
  redisplay;
  sleep 2;
  insert into part (partno, partname, type)
    values (partno, name, type);
  commit;
end
```

Resume

Ends the current operation and positions the cursor as specified.

Syntax

```
resume
resume entry
resume field fieldname | tablefieldname.columnname
resume menu
resume next
resume nextfield
resume previousfield
```

resume

Positions the cursor where it was prior to the current operation

resume entry

Generates a before field activation for the current field, which causes the before field activate block for the current field to be activated (if one was declared). Must be used whenever the application needs to trigger an entry activation.

Not restricted to before activation blocks. Can be used in a frame activation or display submenu. Do not use with a run submenu statement.

resume next

Used within a field or column exit activation. Continues the operation that triggered the activation when issued from inside a field or table-field column activate block). For example, resume next moves the cursor to the next field if a user fills in a field and presses Return.

resume field *fieldname*

Positions the cursor on *fieldname*; either the first space in a simple field or the first column of the first visible row of a table field. The *fieldname* is a 4GL name.

resume field *tablefieldname*.*columnname*

Places the cursor on the specified column within the named table field. *columnname* is a 4GL name. *Tablefieldname* is not a 4GL name.

resume menu

Positions the cursor on the menu line

resume nextfield

Positions the cursor on the next accessible field on the form, and causes any before activation for that field to occur

resume previousfield

Positions the cursor on the previous field on the form that is accessible, and causes any before activation for that field to occur

Description

The 4GL resume statement ends the current operation and positions the cursor in the window as specified. Because the resume statement closes the current operation, it is typically the last statement within an operation specification or within the statement list of an if statement. Statements following a resume statement are not executed.

Follow these guidelines when using the resume statement:

- The resume entry statement causes the FRS to trigger a before field activation for the current field. You cannot use this statement with a run submenu statement, but you can use it from a display submenu statement.
- The resume field statement specifies any field or column on the form.
- The resume menu statement positions the cursor on the menu line.
- The resume next statement causes the operation that triggered the activation to continue; for example, proceeding to the next field. You cannot use this statement with a run submenu statement.
- The resume nextfield statement positions the cursor on the next accessible field on the form.
- The resume previousfield statement positions the cursor on the previous accessible field on the form.

Examples

The first examples are resume field statements. Position cursor on the *name* field:

```
resume field name;
```

Position cursor on the *school* column of the table field called *children*:

```
resume field children.school;
```

Position cursor on the field named in the *resfield* field:

```
resume field :resfield;
```

Position cursor in the column named in the *col* field in the table field called *children*:

```
col = 'school';
resume field children.:col;
```

Positions the cursor on the Empname column in the table Emptbl:

```
resume field emptbl.empname;
```

Positions the cursor where it was prior to the 'Save' operation:

```
'Save' =
begin
  /* Do save */
  if (save_rc != ok) then
    /* Do error processing */
    resume;
  else
    resume field start;
  endif
end
```

Resume menu statement that positions the cursor on the menu line:

```
resume menu;
```

Return

Closes the current frame and returns to the calling frame; also returns a value from a 4GL procedure or frame.

Syntax

```
return [expression]
```

expression

Specifies a single-valued expression. Its data type must be compatible with that of the field to which its value is assigned. Declare its return data type when you define the frame or procedure containing this **return** statement in ABF.

Description

The 4GL return statement returns you to a calling 4GL frame from a called frame or procedure.

In returning from a called frame, return closes the current frame and returns control to the frame that called it with a callframe statement. The calling frame resumes execution at the statement following callframe. If the current frame was the first one when the application started up, the return statement closes the entire application, functioning as an exit statement.

The return statement has the same effect when executed within a submenu as it has within the main menu of a frame. In returning from a called procedure, return passes control back to the calling frame, which resumes execution at the statement following callproc.

The return statement can pass a value back to the frame that called the current frame or procedure. You can use this feature to transmit a return status back to the calling frame. For information on returning a value to the top frame, see Returning to the Top Frame (see page 1337).

The return statement can return only simple data types. For example, you cannot return a record, but you can return a record attribute of a simple datatype.

Example

The **return** operation returns control to the calling frame, passing back the value in the field called Status:

```
'End' =
begin
  return status;
end
```

RewindFile( )

Positions the file to offset 0.

Syntax

```
[returnfield = ] [callproc] rewindfile(handle =
  handle )
```

returnfield

Specifies the name of field to which the result is returned. Integer.

handle

Specifies the file identifier used by the 4GL file access functions. Integer.

Description

The rewindfile() function positions an open file to the beginning (offset 0). The file must have been previously opened with the openfile() function and the file's handle must be known to 4GL.

You can rewind all file types. For file types that allow the positionfile() statement, the rewindfile() statement is the same as the statement:

```
positionfile (handle = handle, offset = 0)
```

You cannot rewind files opened with the append or create modes.

The procedure returns 0 if the function completes without error.

Example

Rewind the file identified by the handle in the variable *fileno*.

```
status = callproc rewindfile(handle = fileno)
```

Run Submenu

Runs a submenu, not directly linked to a form, within a frame.

Syntax

```
run submenu begin | { operations end | }
```

Description

Run submenu displays a submenu that is not directly linked to a displayed form. Use run submenu with an operation list if you want the form to be inactive; the cursor stays on the menu line, and you must make a selection before regaining access to the form. You do not have access to form fields, and field and column activations are not allowed. The submenu is not influenced by form mode. You must include an endloop statement to end the submenu loop.

The run submenu statement signals the start of a submenu that replaces the original menu line for the form. The submenu consists of the double keywords run submenu followed by an operations list (that is, a list of activations and statement blocks). The submenu operation can contain a timeout section (but not a database event section). The keyword end that concludes the run submenu statement must be followed by a semicolon.

As with submenus produced by attached queries, you can include an initialization block in a run submenu statement. This initialization operates the same way as in an initialization statement, except that you cannot declare variables in a run submenu initialization.

Run submenu allows you to display a submenu without attaching it to a query. If your 4GL specification does not require an attached query, do not construct one to display this submenu. 4GL supports two statements, run submenu and display submenu, that display a submenu without an attached query. The principle difference between the two statements is that, in display submenu, the form remains active, while in run submenu, the form is inactive. See Display Submenu (see page 1007).

You cannot use the resume menu or resume entry statement in a run submenu and its operations.

Example

This example runs a submenu with two options:

```
run submenu
begin
    'Select' =
    begin
        ...
    end
    ...
    'End' =
    begin
      endloop;
    end
end;
```

Scroll

Performs a table-field scroll.

Syntax

Target Scroll:

scroll [*formname*] *tablefieldname* to [*record* | end]

Single-Line Scroll:

scroll [*formname*] *tablefieldname* up | down

formname

Specifies the name of the form in which the table field *tablename* is displayed.

Formname is optional in the code for a 4GL frame, because the scroll command applies only to the current form in this context.

Formname is required in a 4GL procedure. It can be expressed as a character string, with or without quotes, or as a program string variable.

tablefieldname

(Required) Specifies the name of the table field to scroll

record

Specifies the number of the record within the data set to which the cursor is to scroll. *record* has a value from 1 up to the number of nondeleted records in the data set.

Description

The scroll statement executes a table-field scroll. The two types of scroll statements are *target* and *single-line* scrolls.

- A *target scroll* scrolls the cursor to a specified record in the table field data set, not a row on the table field displayed in the window. It brings the target record into the table field display and places the cursor on it. It requires the first syntax line shown above.
- A *single-line scroll* performs the same sort of scroll as the upline and downline FRS commands. It scrolls up or down a single row of the table field. The single-line scroll requires the second syntax line.

  When used for a table field with a data set, the single-line scroll simply moves the data set rows up or down one line in the display.
- The scroll down version scrolls the data set record displayed in the bottom row of the table field out of sight. All other records in the table-field display move down one row, and the previous, heretofore invisible, record in the data set (if any) appears as the first row in the display.
- The scroll up version scrolls the top row of the table field out of sight. All other records in the table-field display move up one row, and the next, heretofore invisible, record in the data set (if any) appears as the last row in the display.

When a scroll occurs first, not only do the values for each column scroll, but the change variable associated with each value also scrolls, even out of the display window if necessary.

A row's record number can be determined by means of the _record constant, as described in the unloadtable section in this chapter. A deleted row has a negative record number and cannot be the target of a scroll.

To scroll to the last non-deleted record in the data set, use the keyword **end** in place of the record number. The row to which the program scrolls becomes the current row of the table field. Therefore, if you issue a resume field statement at this point for the table field, the cursor is positioned on that row.

Examples

These operations perform target scrolls to the bottom, top, and middle of the data set on the form Empform:

```
'Bottom'=
begin
  scroll empform employee to end;
end

'Top'=
begin
  scroll empform employee to 1;
end

'Middle'=
begin
  inquire_forms table empform
    (:rows = datarows(employee));
  rows := rows/2;

  if (rows>0) then
    scroll empform employee to :rows;
  endif;
end
```

Note that the form name is not required to scroll the current form. For example, to perform the first target scroll on the current form:

```
scroll employee to end;
```

Find a particular record in the data set:

```
'Find'=
begin
  /* Prompt for name to search for */
  searchname := prompt
    'Lastname to search for: ';
  /* Loop through data set and */
  /* stop when name is found   */
  unloadtable employee (record = _RECORD)
  begin
    if (employee.ename = searchname) then
      /* Scroll to specified record name */
      scroll empform employee to :record;
      resume field employee;
    endif;
  end
  /* All rows in data set searched   */
  /* and name not found.        */
  message 'Cannot find named employee.';
  sleep 2;
end
```

Select

Retrieves rows from one or more database tables.

Syntax

The select statement has four variants:

- Select to a complex object
- The Master/Detail query
- Select loop
- Select to simple objects

The syntax for each variant follows.

Select to a complex object:

```
complexobjectname := [repeated] select
  [all|distinct]
  [fieldname =] expression | [tablename.]columnname
  {, [fieldname =] expression |
    [tablename.]columnname}
  from fromsource {, fromsource }
  [where qual]
  [group by columnname{, columnname} [having qual] ]
  [order by orderfieldname [sortorder]
    {, orderfieldname [sortorder] } ]
[begin | {
  submenu
end | }]
```

Master/Detail query:

```
formname := select [all | distinct]
  [fieldname =] [tablename.]columnname
  {, [fieldname =] [tablename.]columnname}
  from fromsource {, fromsource }
  [where qual ]
  [group by columnname{, columnname} [having qual] ]
  [order by orderfieldname [sortorder]
    {, orderfieldname [sortorder] } ]
tablefieldname = select [distinct]
  [fieldname =] [tablename.]columnname
  {, [fieldname =] [tablename.]columnname}
  from fromsource {, fromsource }
  where qual
  [group by columnname{, columnname} [having qual] ]
  [order by orderfieldname [sortorder]
    {, orderfieldname [sortorder] }]
[begin | {
  submenu
end | }]
```

Select loop:

```
[repeated] select [all | distinct]
  [ [:]simpleobjectname =] expression |
  [tablename.]columnname
  {, [ [:]simpleobjectname =] expression |
  [tablename.]columnname }
from fromsource {, fromsource }
[where qual]
[group by columnname {, columnname} [having qual] ]
[order by orderfieldname [sortorder]
    {, orderfieldname [sortorder] } ]
begin | {
  statementlist
end | }
```

Select to simple objects:

```
[repeated] select [all | distinct]
  [ [:]simpleobjectname =] expression |
  [tablename.]columnname
  {, [ [:]simpleobjectname =] expression |
  [tablename.]columnname }
from fromsource {, fromsource }
[where qual]
[group by columnname {, columnname} [having qual] ]
[order by orderfieldname [sortorder]
    {, orderfieldname [sortorder] } ]
```

complexobjectname

Specifies the name of a form, table field, table field row, record, or array to which you are assigning values. Where the table-field name and form name are identical, specify the *complexobjectname* as *formname.tablefieldname*.

The data types of *objectname* and *tablename.columnname* or *columnname* must be compatible.

The current row of a table field can be written as *tablefieldname[].*

fieldname

Specifies the name of the object that receives the value from the specified database column:

- For a form, the name of a simple field
- For a table field or a row of a table field, a column in a table field
- For a record or array, an attribute of the record

expression

Specifies any legal 4GL expression whose type is compatible with *simpleobjectname* or *fieldname* to which *expression* is being assigned.

If the select statement refers to an *expression* rather than a *columnname*, the syntax is as follows:

- For a select into a complex object, the [fieldname **=**] preceding expression is required.
- For a *select loop* or *select* to a *simple object*, the [[**:**]*simpleobjectname* **=**] preceding *expression* is required, unless the expression is an asterisk (*) and all columns in the table correspond to fields on the form.

[*tablename*.]*columnname*

Specifies the name of the source column in the database table or tables from which data is selected. [*tablename.*] specifies the table containing the source column. It can be omitted if only one table in the from clause contains the source column. If the from clause specifies a corrname following the tablename, you must specify *corrname.columnname* instead of *tablename.columnname.*

fromsource

Specifies the name of the source of the columns from which data is selected. A source can be a table or an outer join between tables. A table is specified with the syntax:

```
[owner.]tablename [corrname]
```

A join between tables is specified with the syntax:

```
source join_type join source on search_condition.
```

where a *source* can be a table name or a join. For more information on the from clause, see the section, The From Clause.

qual

Specifies a logical expression of conditions that all rows selected must meet. It cannot include 4GL names. The qualification function is allowed as a condition. See The Qualification Function.

columnname

Specifies the column in the group by clause

orderfieldname

Specifies the name of a column on which to sort a 4GL name.

For a *select into a complex object* or *master/detail query, orderfieldname* must match:

- The fieldname in a fieldname **=**expression clause
- The fieldname in a fieldname **=** [tablename.]columnname clause
- The columnname in a [tablename.]columnname clause

For a *select loop* or *select into simple objects,* similar rules apply, except that the matching *fieldname* must be a simple field or simple variable, and can be preceded by a colon.

If more than one orderfieldname appears in an order by clause, the retrieved rows are sorted on the basis of the first, then on the second within the results of the first, and so on.

sortorder

Indicates whether the preceding *orderfieldname* is to be sorted in ascending (asc) or descending (desc) order. If omitted, asc is assumed. This is a 4GL name.

simpleobjectname

Specifies the name of a simple field, a local or global simple variable, a table-field cell, or a simple record attribute. The data types of the value and the object must be compatible.

Write table-field cells as :*tablefieldname***.***columnname* (which refers to the row under which the cursor is positioned) or
*:tablefieldname***[***integerexpression***].***columnname*

Write the names of record attributes as :recordname.attributename
or
:arrayname**[**integerexpression**]**.attributename

Where the table field name and the form name are identical, write the table field as *formname.tablename.*

If you specify a table-field cell or record attribute, a preceding colon is mandatory.

submenu

Specifies a separate menu displayed for the duration of a complex structure retrieval. It allows you to exercise some options with respect to the data displayed, including displaying the next row of data. Separate the *submenu* from the rest of the select statement by braces or by a begin and end pair.

Description

The 4GL select statement has the following variants:

- *Select to a complex object* assigns values from rows in a database table to a form, table field, table field row, record, or array. You can combine this with an assignment statement in a variety of query formats to perform different kinds of assignments to complex structures.

  Attached queries (discussed below) can be nested to any level.

  Submenus can be used with attached queries. They enable you to display the next master row in the data set, or to perform an operation on the current row.
- *Master/detail query* is a form of the attached query that selects data into the simple fields of a form (the master row) and also into a table field (the detail rows), then displays a submenu of operations that can be carried out for each master row selected.
- *Select loop* iterates through the rows of a query, processing them one row at a time. Each column of the row can be assigned to a simple field, simple variable, table field cell, or record attribute. This allows you to perform a sequence of 4GL statements for each row or record. More information on select loops is given in the section Using a Select Loop.
- *Select to simple objects* assigns values from a single row in a database table to various fields and/or simple variables.

When any one of these queries is executed, the query runs to completion, and the matching rows are stored in a temporary data set. The select statement then accesses the data from the data set.

For maximum performance when your code executes a particular query several times, use the reserved word repeated. Use the reserved word distinct to eliminate duplicate rows.

Limitations to the select statement:

- If the select statement whose target object is not a form returns no rows or records, it is unspecified whether the target object is cleared.
- Only the table's owner and users with select permission can select from a table.

If a where clause is included, the select statement returns values from the specified database table columns for all rows satisfying the where clause. The qualification function can be used in the where clause. You can store the where clause in a variable. However, in this case, you cannot use the qualification function.

Use the optional order by clause to sort rows alphabetically or numerically, in ascending or descending order.

See your query language reference guide for information on the use of the reserved words all, from, group by, and having.

Overview of the Select Statement

To read data from database tables into forms, fields, and variables, use the select statement. The select statement takes values from the columns in the specified database table for all rows that satisfy the condition specified in an optional where clause.

Any field can be the target of a select query. 4GL allows you to select into a simple field, a table field, a table field row, a table field cell, record, array, an attribute of an array, or an attribute of a record.

The select statement can contain several clauses: the select clause, the from clause, and the optional where, order by and union clauses. The where clause can contain a further optional qualification clause. The following shows a simplified syntax for select:

```
objectname = select
    [fieldname = ] [tablename.]columnname
    {, [fieldname = ] [tablename.]columnname ]
  from tablename
  [where qual ]
  [order by orderlist];
```

Select syntax including the union clause is shown in The Union Clause. When field names in different tables are the same, use *tablename.columnname* format to identify the source tables.

This example of a select statement contains a from and a where clause:

```
deptform := select lname, fname
    from personnel
    where empnum = 7;
```

In this example, Lname and Fname are columns in the Personnel table. The form Deptform has similarly-named simple fields. Values returned for Lname and Fname are displayed in the corresponding simple fields. The query retrieves the first and last name of employee number 7.

The select statement can contain the following clauses:

- The select clause names the database columns from which you want to retrieve data. In the example, Lname and Fname are the names of the columns.
- The from clause specifies the database tables to which the columns belong. In the example, this is the Personnel table.
- The where clause allows you to fine-tune your query by specifying conditions you want the selected columns to meet. This is useful for a simple-field assignment. In the example, the where clause limits the query results to the name of employee number 7. For more details, see The Where Clause.

  To retrieve many rows (as in a table-field or array assignment), use a where clause that specifies several rows. This example retrieves employees who make more than $20,000:

  ```
  perstable := select lname, fname, salary
    from personnel
    where salary >= 20000;
  ```

  The where clause is optional, except in the detail portion of Master/Detail query.

  You can use the optional qualification function in a where clause. See The Qualification Function.

- The optional order by clause allows you to specify sorting criteria when a query returns more than one value. The following example retrieves more than one row; the statement orders them by the named column, Lname.

  ```
  depttable := select lname, fname
    from personnel
    where emptitle = manager
    order by lname;
  ```

  When you sort, you must specify field names from the complex object into which you are selecting. These are the same as column names in the database table.

- The optional union clause combines the results of a number of select statements. For more information, see The Union Clause.

From Clause

The from clause specifies the source tables from which data is to be read. The specified tables must exist at the time the query is issued. The *from_source* parameter can be:

- One or more tables, specified using the following syntax:

  [*owner*.]*table* [*corr_name*]

  where *table* is the name of a table, *owner* is the name of the user that owns the table, and *corr_name* is a correlation name for the table. A correlation name is an alternate name for the table, often used to abbreviate long table names. If you assign a correlation name to a table, you must refer to the table using the correlation name. The table name can be a 4GL name; however, if a 4GL variable is used, an owner cannot be specified, unless the owner name is included as part of the value. In other words, the variable can contain the value *owner*.*tablename*.

- A join between two or more tables, specified using the following syntax:

  source join_type join source on search_condition

  where:

  - The *source* parameter is the table or outer join where the data for the left or right side of the join originates.
  - The *join_type* parameter specifies inner, left, right, or full outer join. The default join type is inner.
  - The *search_condition* is a valid restriction, subject to the rules for the where clause. The search condition must not include aggregate functions or subselects.

For example:

```
select e.ename, d.dname from
    (employees e left join departments d
      on e.edept = d.ddept);
```

You can specify a maximum of 126 tables in a query, including the tables in the from list and tables in subqueries.

See the *SQL Reference Guide* for more information about outer joins.

Where Clause

The optional where clause qualifies a database retrieval according to the specified logical (Boolean) expressions. The where clause is simply a Boolean combination of conditions. A condition can be an SQL predicate (for example, [*tablename.*]*columnname* = *expression*) or it can be an invocation of the qualification function.

You can use a simple logical expression, which compares a column in a database table to an expression, such as a constant value or a field in the current form. For example:

```
empform := select name, jobtitle
  from employee
  where salary >= :sallimit;
```

This statement selects into Empform the names and job titles of all employees whose salaries are equal to or greater than the value entered in the Sallimit field.

In SQL, you can use the underscore (_) and the percent sign (%) as wildcard characters. The underscore (_) represents a single character, and the percent sign (%) represents any number of characters. The available wildcard characters depend on the installation parameters of the II_PATTERN_MATCH logical/environment variable. For a complete list of comparison operators, see Using Logical Expressions.

The II_PATTERN_MATCH logical/environment variable sets pattern matching to SQL or QUEL. Unless II_PATTERN_MATCH is set, you must use the wildcard characters for the query language of the application. By default, II_PATTERN_MATCH specifies QUEL pattern-matching characters, which are different from those in SQL. For more information, see Notes for Users of QUEL (see page 1263).

You can use a 4GL string variable to specify the qualification for the where clause. For example, you can declare a variable "whclause" and assign as its qualification the following value:

```
whclause = 'emp.salary > 40000';
```

You then can use this variable as the qualification for the where clause in your **select** statement, as follows:

```
select * from emp
  where :whclause;
```

Queries in which the where clause is specified in a variable are not suitable for use as repeated queries. The keyword repeated is ignored if you specify it for such queries.

When using variables or constants in select statements in ABF, any variable or constant on the left side of a comparison operator is treated as a column name. Any variable or constant on the right side is treated as a value.

For example, in the following query:

```
select * from :a where :b = :c
```

:a is treated as a table name, :b is treated as a column, and :c is treated as a value (and therefore may have quotes surrounding it depending on the data type).

The entire where clause of a query can be put into a single variable (for example, where :a).

Note: The phrase ":a between x and y" is treated as a column name, as is ":a = (select ....)". In those cases, if you wish to use :a as a value, either the query must be rewritten (for example, mycolumn >= :a and mycolumn <= :a) or an override can be used to force all parameters in the where or on clauses to be used as values (and thus have quotes where required). To set this override, issue the command **ingsetenv II_ABF_ASSUME_VALUE Y** or set II_ABF_ASSUME_VALUE to Y in the environment where the ABF application is compiled.

Qualification Function

The qualification function, when included in a where clause, allows you to set up the exact search criteria at run time in the specified simple fields or table field columns of the current form. The qualification function interprets the values in these fields as expressions, and builds the where clause on these. However, the qualification function cannot be used when the where clause is specified in a variable, as described above.

Placing the optional qualification function in a where clause allows the end user to include comparison operators (such as != or <) and wildcard characters at run time in the search condition. The end user can type values, comparison operators, or wildcard characters into the visible simple fields or table field columns in the current form specified by the qualification function.

The requirements for using the qualification function are:

- The current form must be in Query mode.
- In a qualification function, only simple fields or table field columns on a form (not local variables) are allowed.
- Any table field on which you allow qualifications must be in query mode.

When the end user enters query qualifications in a table field, the where clause is generated as follows:

- Qualifications in table field columns of a single row are joined by AND
- Qualifications in multiple table field rows are joined by OR

The example below shows a query operation that utilizes the qualification function on simple fields of a form:

```
empform := select idnum = empnum, jobtitle,
  salary = empsal
  from employee
  where qualification (jobtitle = jobtitle,
    empsal = salary);
```

By entering the values "%Manager%" and "<30000" in the *jobtitle* and *salary* fields, respectively, you can retrieve information on all employees having a job title containing the word "Manager," and whose salaries are under $30,000.

Without the qualification function in the original example, the where clause searches for a literal match to everything you enter in the field, including the operators > and <. If you enter no values, no restrictions are placed on the retrieval. In this case, a validation error results.

In the select example below, the order number and customer name correspond respectively to the *orderno* and *custname* fields on the *addorders* form:

```
addorders := select orders.orderno,
  customer.custname
  from orders, customer
  where qualification
    (orders.orderno = orderno,
    customer.custname = custname) and
    orders.custno = customer.custno;
```

If you enter ">1000" in the *orderno* field and "A%" in the *custname* field, the query retrieves all customers whose names start with "A" and whose order number is greater than "1,000." If you leave blank a field named in the qualification function, no qualification is generated for that field.

The following example shows a query operation that utilizes the qualification function on columns of a table field:

```
emptf := select empnum, jobtitle, empsal
from employee
where qualification
(jobtitle = emptf.jobtitle,
 empsal = emptf.empsal);
```

Queries in which the qualification function is used are not suitable for use as repeated queries. The keyword repeated is ignored if you use it with such queries.

Order By Clause

If more than one *orderfieldname* is specified in the order by clause, rows are ordered on the basis of the first *orderfieldname*, second *orderfieldname*, and so on, and within values of that field, on the basis of the specified *sortorder*. The syntax for the order by clause is as follows:

```
order by orderfieldname [sortorder]
  {, orderfieldname [sortorder]}
```

Specify a *sortorder* of asc for ascending (the default), or desc for descending.

Note that you must specify the *result* names in the order by clause in any select statement. Because 4GL gives the field name as the result name for columns and target expressions, you must sort by the field names in 4GL, as in the following example:

```
empform := select ename, dept
  from employee
  order by ename;
```

When you are selecting to a different name, order by the *result* name:

```
result = columnname
```

or

```
columnname as result
```

In this example, the query sorts by the result (field) name, *person*, and not by the column name Ename.

```
newform := select person=ename, dept
  from employee
  order by person;
```

Union Clause

Use the union clause to combine the results of a number of select statements. The syntax for this clause was not shown in the main syntax for the select statement to save space. The general syntax is:

```
objectname :=subselect
union [all] subselect {union [all] subselect}
```

Here *subselect* is a select statement without the repeated keyword or the order by clause. The *subselect* can optionally be enclosed in parentheses. Any number of subselects can be joined together with the union clause.

The names in the column list for the first *subselect* build the correspondence to the form fields or variables. A name is also a literal. All result rows of any select statement in a union must have the same data type. However, the column names do not have to be the same. The following select statement selects the names and numbers of people from different organizations:

```
empform := select ename, enumber
    from employee
    union
    select dname, dnumber
    from directors
    where dnumber <= 100;
```

A *subselect* refers to an expression or a form field instead of a column in the database table. In this case, you must use the following syntax for the name in the column list:

```
fieldname = expression | form_field
```

For example:

```
empform := select ename, compens = esal + ebens
    from employee
union
    select dname, compens = dsal + dbens
    from directors;
```

or

```
empform := select ename, compens = esal + ebens
    from employee
union
    select dname, compens = :total_benefits
    from directors;
```

Query Targets

The target of a select query can be of any Ingres data type, a field or variable, form, table field column or row, record, record attribute, array or array record attribute. When a query fetches one of these, values can be assigned directly to it.

The following example assigns one value from *dbexpr* into the *col* column of the current row of the *tbl_fld* table field:

```
form = select :tbl_fld.col = dbexpr from dbtable
```

You can assign values to any of a displayed row's columns by using a row reference. For example:

```
tbl_fld[ 2 ] = select col = dbexpr from dbtable ;
```

```
array[ i ] = select * from dbtable ;
```

The data types of the target object and the database column from which the data is selected must be compatible. If they have the same name, do not specify *fieldname*.

Use the reserved word all in place of the target list in any select statement to assign values to all the simple fields in a form or to all the visible columns of a table field. Each field or column in the form must correspond to a database column with the same name and data type. The database can contain columns that do not match those in the form, but the form cannot contain unmatched simple fields or table field columns. Append the reserved word all to the tablename, as shown below:

```
form := select tablename.all
```

You can also use the asterisk (*) in any select statement to assign values to all the visible simple fields of a form or to all the visible columns of a table field, or to all the attributes of a record or array record. The asterisk replaces the list of column names, as shown below:

```
deptform := select * from deptable;
```

4GL recognizes complex components, including arrays and record types, in most contexts where it recognizes forms and table fields. Like forms and table fields, they cannot be assigned or compared directly in any way. This also means that complex components can be query targets. For example:

```
record = select * from desc where ... ;
```

Selection into a complex data component is similar to selecting into table fields in that the data is cleared (default values are displayed) before the query is executed. This is different from form selects, which do not change the values of any fields not specified in the target list.

See Selecting into Records in Selecting into Table Fields (see page 1108) for information on selecting into a record without clearing the current values.

Selecting into Table Fields

You can use several types of select statements with table fields. If the table field is in query mode, after the select is executed the table field is changed to update mode.

The following examples illustrate the use of select statements with table fields:

tablefield[] = **select * from** *tablename*

Selects into the current table field row. For example:

```
emptbl[] = select * from emp;
```

tablefield[] = **select** *col*, *col* **from** *tablename*

Selects into the columns, including hidden columns, of the current table field row. Columns not assigned are cleared. For example:

```
emptbl[] = select name, oldname =
  name from emp;
```

tablefield[*n*] = **select * from** *tablename*

Selects into the indexed table field row. The row *n* must currently contain data (_state**=** 1 or more). For example:

```
emptbl[3] = select * from emp
  where name = 'Jones';
```

tablefield[*n*] = **select** *col*, *col* **from** *tablename*

Selects into the columns, including hidden columns, of the indexed table field row. Columns not assigned are cleared. The row *n* must currently contain data (_state**=** 1 or more). For example:

```
emptbl[3] = select name, salary
  from emp
  where name = 'Smith';
```

select :*tablefield.col1* = *col2* **from** *tablename*

Selects into the columns, including hidden columns, of the current table field row. Columns not assigned are not changed. For example:

```
select :emptbl.salary = salary
  from emp
  where name = :emptbl.name;
```

select :*tablefield*[*n*].*col1* = *col1* **from** *tablename*

Selects into the columns, including hidden columns, of the indexed table field row. Columns not assigned are not changed. The row *n* must currently contain data (_state**=** 1 or more). For example:

```
select :emptbl[5].salary = salary
  from emp
  where name = 'Jones';
```

select :*tablefield.col1* = *col1*, *col2* **from** *tablename*

Selects into the columns, including hidden columns, of the current table field row and simple field or variable with one query. For example:

```
select :emptbl.salary = salary, name
  from emp
  where name = 'Smith';
```

select :*tablefield*[*n*].*col1* = *col1, col2* **from** *tablename*

Selects into the columns, including hidden columns, of the indexed table field row and simple field or variable with one query. The row *n* must currently contain data (_state**=** 1 or more). For example:

```
select :emptbl[5].salary = salary, dept
  from emp
  where name = 'Jones';
```

Selecting into Records

The following types of select statements are available for records:

record = **select * from** *tablename*

Selects from the columns of a database table into record attributes. Every attribute must have a corresponding column with the same name and data type in the table. Default values (blanks or zeroes) are loaded into record attributes for which there is no column of the same name, or whose data type is not coercible from the data type of the corresponding column.

The following example selects from the columns of the Employee table into the record *emp_rec*:

```
emp_rec = select * from Employee
```

record = **select** *col* [, *col*] **from** *tablename*

Selects from specified columns of a database table into corresponding record attributes. The contents of other attributes are not changed.

The following example selects from the Name and Job_title columns of the Employee table into the corresponding attributes of the record *emp_rec*:

```
emprec = select name, job_title
      from Employee
```

select :*rec*.*attribute* = *col* [,:*rec*.*attribute* = *col*] **from** *tablename*

Selects from the column of a database table into a selected record attribute. The contents of other attributes are not changed. You must use a colon before the record name.

The following example selects from the Job_title column of the Employee table into the *title* attribute of the record *emp_rec.*

```
select :emprec.title = job_title from Employee
```

select :*rec*.*rec*.*attribute1* = *col* [,:*rec*.*rec*.*attribute* = *col*] **from** *tablename*

Selects from a column of a database table into the attribute of a nested record (that is, a record that is an attribute of another record). The contents of other attributes are not changed. You must use a colon before the record name.

The following example selects from the City column of the Employee table into the *city* attribute of the *address* record. *Address* is an attribute of the record *emp_rec*:

```
select :emprec.address.city = city
     from Employee
```

You can use the where and order by clauses with these select types.

Selecting into Arrays

To select into an array, use the following syntax:

```
array = select col1, col2 from tablename where ...;
```

If the query is executed successfully (even if no rows are returned), then previously allocated records are removed from the array before the retrieved rows are assigned to it, just as they are with table fields.

If you want to retrieve records into an array without clearing existing values, use a select loop to assign values to individual attributes in the array, as follows:

```
i = maxrow + 1 ;
select h_col1 = col1 [,h_col2 = col2] from tablename where ...
begin
  array[i].attr1 = h_col1;
  [array[i].attr2 = h_col2;]
  i = i +1 ;
end
```

You must define *h_col1*, *h_col2*, and so on, as local variables of the appropriate data type. Each iteration of the select loop automatically appends a new record after the last record of the array. You receive a runtime error if you attempt to create empty records by increasing the index number (*i*) by any value greater than 1.

Note: Do not attempt to select a large number of rows into a table field or array because of memory limitations inherent in all computer systems.

Select to a Complex Object

The select variant *select to a complex object* is always associated with a query assignment statement. The select statement is a database expression, because it is equivalent to the values it selects. The syntax for this variant is shown in the main syntax section for the select statement above.

The select statement makes up the right side of an assignment statement, while the name of the form, table field, table-field row, record, or array to which the selected values are assigned appears on the left side. If the select into a form returns no rows, then the previous contents of the form components into which you are retrieving the data do not change.

The select variant select to a complex object can be further subdivided into three types:

- The singleton query
- The table field or array query
- The simple attached query

These are discussed in the following sections.

Singleton Query

A *singleton query* gets values from the first (or only) row in the data set retrieved from the database table and places these values in simple variables and simple fields of the current form, columns of a row of a table field, or attributes of a record, without the use of a submenu. The query stops after the first row, and only one row is returned.

The following singleton query selects a row of values from the Personnel table into a form named Deptform:

```
deptform := select lname, fname
from personnel where empnum = 7;
```

The following singleton query selects into a record in an array:

```
deptarray[1] := select *
from personnel where empnum = 7;
```

This example selects the same row of values as in the previous example, but assigns them to the first record in the array *deptarray*. This example assumes that all attributes of *deptarray* have corresponding columns with the same names in the Personnel table.

Select to a Table Field or Array

A *table-field query* or *array query* selects rows of a table from the database into a table field on a form or into an array, without the use of a submenu.

In a table-field query, the maximum number of rows that fit within the table field are displayed, and the remainder are held in a buffer (the underlying data set) by the FRS. You can view all the rows selected by scrolling through the table field.

The following table-field query retrieves values from the *projects* table into a table field. Columns in the *emptable* table field correspond to columns in the Projects table.

```
emptable := select project, hours
  from projects
  where empnum = :empnum;
```

The following array query retrieves values from the Projects table into an array. Attributes in the *emparray* correspond to columns in the Projects table.

```
emparray := select projects, hours
  from projects
  where empnum := empnum;
```

Simple Attached Query and Submenu

A *simple attached query* selects several rows of data to a form, table-field row, or record. When used with a form, a *simple attached query* performs these functions:

- It *attaches* the as-yet-undisplayed rows to the form.
- It displays the first row in the simple fields of the form and uses a submenu to provide access to each of the subsequent rows.
- It displays a submenu of operations that can be carried out for each returned row that is displayed.
- It provides a submenu operation that invokes a next statement for displaying subsequent rows.

Submenus are discussed in more detail later in this section. Refer also to the section Next. An example of a simple attached query is shown below:

```
'Find' =
begin
  editparts := select partname = partname,
    partno = partno, descr = descr,
    cost = cost
    from part
    where cost < :mincost
  begin
    'Change' =
    begin
      /* code to replace the database entry
      for the part with the new values */
      next;
    end
    'Delete' =
    begin
      /* code to delete the displayed part
      from the database */
      next;
    end
    'IncreasePrice' =
    begin
      cost := cost * 1.1;
    end
    'Next' =
    begin
      next;
    end
    'End' =
    begin
      endloop;
    end
  end;
  message 'Done with query';
  sleep 2;
  clear field all;
end
```

When you choose Find, the application displays on the form the first row selected, along with this submenu:

```
Change Delete IncreasePrice Next End
```

A single select statement starts the database retrieval and displays the new list of operations. 4GL combines these two actions within the select statement because they are typically tied together in forms-based applications.

You can determine the number of database rows processed by the attached query through the use of the rowcount reserved word in an inquire_sql statement following a database access. The example below assigns the number of rows accessed to the field called rows:

```
inquire_sql (rows = rowcount);
```

The rowcount constant is undefined while the submenu is active. After the submenu closes, rowcount equals the first row plus the number of rows displayed with a next statement. After a singleton query, rowcount has a value of 1.

Using attached queries with table-field rows and records is similar to using them with forms, except that the rows are assigned to table-field columns and attributes of a record instead of fields of a form.

Master/Detail Query

A *Master/Detail query* is a form of the attached query that selects data into the simple fields of a form (the master row) and also into a table field (the detail rows), then displays a submenu of operations that can be carried out for each master row selected. The first statement is the *parent* or *master* query; the second is the *child* or *detail* query.

Inclusion of a submenu offers you a choice of operations that can be carried out for each master row selected, and provides the means for displaying the next master row. If you omit the submenu, a single master row and all the details for that master are displayed at once; there is no way to see the next master row.

The Master/Detail query requires this syntax:

```
formname := select [distinct]
  [fieldname =] [tablename.]columnname
  {, [fieldname =] [tablename.]columnname}
  from tablename [corrname] {, tablename [corrname]}
  [where qual ]
  [group by columnname {, columnname} [having qual] ]
  [order by orderfieldname [sortorder]
    {, orderfieldname [sortorder] } ]
tablefieldname = select [distinct]
  [fieldname =] [tablename.]columnname
  {, [fieldname =] [tablename.]columnname}
  from tablename [corrname]{, tablename [corrname]}
  where qual
  [group by columnname{, columnname} [having qual] ]
  [order by orderfieldname [sortorder]
    {, orderfieldname [sortorder] } ]
[begin | {
  submenu
end | }]
```

This syntax differs from that of a simple attached query in the following ways:

- Two queries are specified rather than one.
- Only two assignment statements are allowed, and each must be to a complex structure.
- No semicolon is allowed between the two assignment statements.
- If the master query has no where, group by, or order by clause, then the last tablename in the from clause must be followed by a *corrname*. You can specify *corrname* to be the same as the *tablename*.

An example of a Master/Detail query is shown below:

```
'Find' = begin
  empinfo := select empname, title,
    hourly_rate
    from staff s
    order by empname
  tasktable := select task, hours
    from tasks
    where tasks.name = :empname
    order by task
  begin
    'Cancel' = begin
    /* code to cancel the task */
      next;
    end
    'Update Hours' = begin
    /*code to change the hours on a task */
    end
    'Next' = begin
      next;
    end
    'End' = begin
      endloop;
    end
  end;
  commit;
  clear field all;
end
```

In this example, the first query selects the master rows into a temporary data set and displays a single row in simple fields on the Empinfo form. The second query displays the details for each master in the *tasktable* table field. Once the master and detail rows are displayed, Ingres displays this submenu in the window:

```
Cancel Update Hours Next End
```

Choosing End returns the application to the previous menu. Choosing Next causes the next statement to be executed. Each time the next statement steps forward to the next master record, the second query runs, based on the value from the *empname* column of the new master.

Note that no semicolon is used between the two select statements in the query on the Master/Detail join, or between the second query statement and begin statement (or starting brace) of the submenu.

Rowcount for Master/Detail attached queries is similar to row count for attached queries that place values only in simple fields. Rowcount is undefined while the submenu is active.

After the submenu is closed, rowcount returns the number of master rows you are shown (the first master row plus the number of master rows displayed with the next statement). You can use the inquire_forms table to find the number of rows in the table field, and thus determine the number of detail rows for a master.

Submenus

A *submenu* is a separate menu of choices that appears following the display of each master row in the retrieved data set for an attached query (either a simple attached query or a Master/Detail query). Choosing an operation from the submenu causes the designated action to be performed, such as incrementing a salary field or displaying the next master row. Submenu operations always apply to the master row.

When a query with a submenu is executed, the query runs to completion, and a data set is created in a temporary file to hold the selected rows. A submenu appears at the bottom of the frame. Each next statement (associated with an item on the submenu) causes the next row in the data set to be displayed. If this is a Master/Detail query, the master query is performed as above, and another query runs to completion to retrieve the matching detail rows for each master row.

The syntax used to create a submenu is:

```
objectname := select statement
[objectname := select statement]
begin | {
  initialize = begin | {
statements    /* statements executed before first row */
              /* is displayed */
end | }
  "menuitem1" = begin | {
code      /* code for menuitem1 operation */
end | }
"menuitem2" = begin | {
code      /* code for menuitem2 operation */
  end | }
end | }
```

The submenu is separated from the rest of the select statement by braces or by a begin and end pair. In the syntax of a Master/Detail query, the submenu appears after both assignment statements.

Submenus can contain all the same components as main menus. Generally at least one of the menu operations must contain the 4GL next statement, which causes the next row selected to display.

You stay within the submenu until either (1) a next statement is executed after all the rows have been displayed, or (2) an endloop statement is executed. When either situation occurs, the frame's main menu is redisplayed, and control returns to the statement immediately following the assignment statement that contained the submenu.

A submenu is only displayed if the query retrieves rows. The submenu remains active after a resume or a failed validation with the validate statement.

The following example of a simple attached query selects a series of complex values from the database and creates a submenu for the current form:

```
empform := select lname, age, empnum
  from empnum
  where empnum <=10
begin
  initialize =
  begin
    /* Statements here are executed once */
    /* before the first row is displayed */
    message 'Now within submenu';
    sleep 3;
  end

  field 'lname' =
  begin
    message 'You just entered value: ' +
      lname;
    sleep 3;
    resume next;
  end
  'Increment' =
  begin
    age := age + 1;
  end
  'NextRow' =


  begin
    next;
  end
  'End' =
  begin
    endloop;
  end
end;
```

When you choose an operation from the main menu containing this assignment statement, the following submenu appears at the bottom of the frame:

```
Increment  NextRow  End
```

The first row retrieved is displayed in the three fields specified. Each time you choose the NextRow operation from the submenu, the next row is displayed. When there are no more rows, the message "No rows selected" appears at the bottom of the window, and you exit from the current submenu. The main menu for the frame reappears.

If no rows in the database satisfy the query, the message "No rows retrieved" appears at the bottom of the frame, and control transfers to the statements immediately following the assignment containing the submenu.

The submenu definition can contain any legal operations, including menu, key, and field activations.

The first submenu operation definition can be an initialize block. Unlike the initialize block for a frame, the initialize block in an attached query cannot contain variable declarations. The initialize block for the attached query is executed only once, after the master row and matching details have been selected from the database and before their values have been displayed. Both 4GL and SQL statements appears in the initialize block.

The submenu initialization can be used to change the from mode. The default mode is Update. When you exit the submenu, you return to the mode that was in effect when the submenu was invoked, regardless of whether this initial mode was set by the set_forms or mode statement.

The code in the submenu can contain any legal 4GL code. This includes query statements, such as update and delete, as well as other assignment statements involving selects. A return or exit statement within the submenu has the same effect as a corresponding statement from the main menu of the frame. The resume and validate statements operate in the same manner as they do in the main menu; however, if they close the current operation, the submenu is still displayed.

While the submenu is running, the value of the rowcount FRS constant is undefined. Rowcount only has a value when the submenu ends. The single exception is when a query statement in the submenu code is executed; then rowcount is set for that query statement.

Note that when the assignment statement containing the submenu is executed, ABF displays the current form in update mode. You can change this mode in the initialize section of the submenu. When the end user exits from the submenu, the display mode in effect prior to the submenu again takes effect.

Queries with a submenu, like any other query, are always part of a transaction. Shared locks are held on the selected data until the next commit or rollback statement.

You can override this default behavior and cause no locks to be held on the selected data by one of the following methods:

- If your application has previously issued set autocommit on, then a commit is automatically performed after every successfully executed query.
- If you have turned off read locks with the set lockmode statement on the table from which you are selecting, then no locks are held on the table. However, a transaction is still in progress.

It is possible to use a submenu without using an attached query. For this purpose, 4GL provides the run submenu and display submenu statements. See the appropriate sections in this chapter.

Nesting Attached Queries

4GL provides the capability to nest attached queries to any level. Each nested query is placed within the submenu of the higher-level query. Thus, the following syntax is valid:

```
formobject := database_expression
formobject := database_expression
{
  "menuitem1" = {
    formobject := database_expression
    formobject := database_expression
    {submenu}
  }
};
```

In nested queries, if the from clause is the last clause in a master query, then the last table name in the from clause must be followed by a correlation variable (*corrname* in the syntax diagrams for select).

For nested queries with a submenu, the value of the rowcount FRS constant is undefined while the submenu is running. Rowcount only has a value when the submenu ends. The single exception is when a query statement in the submenu code is executed; then the rowcount is set for that query statement.

Using a Select Loop

The *select loop* allows the program to perform a computation for each selected row, rather than to display results to you, as in the *select to a form, table field, record, or array* statement. You can use a select loop to select into a field or variable, a table field, record, an array, or the attributes of a record or array.

A select loop selects and assigns multiple rows of data, one row at a time. A series of specified operations is performed once on the values in each row selected. The syntax for a select loop is as follows:

```
[repeated] select [distinct]
  [ [:]simpleobjectname =] expression |
  [tablename.]columnname
  {, [ [:]simpleobjectname =] expression |
  [tablename.]columnname }
from tablename [corrname] {, tablename [corrname] }
[where qual]
[group by columnname {, columnname} [having qual] ]
[order by orderfieldname [sortorder]
    {, orderfieldname [sortorder] } ]
begin | {
  statementlist
end | }
```

This form of the statement executes the commands in the *statementlist* once for each row returned by the query.

The value selected from the right-hand side of the target list element is assigned to the variable in the left-hand side of the target list element.

- The left-hand side, if present, must be a legal 4GL variable reference. This variable can be a simple field or local or global variable of an Ingres data type, a table field cell, or a record attribute. This allows values to be assigned directly into these fields, simple variables, or records.
- If the left-hand side is not present, the right-hand side must be a database *table.column* reference, in which case the column's name must be the same as a simple field or local variable to which the value is assigned.

During execution of the loop, the value of inquire_sql (rowcount) is not defined; if a query statement is executed within the loop, then rowcount is defined for that statement. After the select loop, rowcount equals the number of rows through which the program looped.

Select loops perform an implicit next operation each time the loop is executed. In addition, an explicit next statement within the body of the loop causes the next row of the data to be selected. Statement execution continues with the statement immediately following next. The loop terminates at the end of the data set. Use the endloop statement to terminate the loop early. For more information, see Endloop (see page 1008).

Select to Simple Objects

Select to simple objects allows you to retrieve a single row from a database table and place the columns of that row into specified simple fields, simple variables, table field cells, or record attributes. The syntax for a select to a simple object is as follows:

```
[repeated] select [distinct]
  [ [:]simpleobjectname =] expression |
  [tablename.]columnname
  {, [ [:]simpleobjectname =] expression |
  [tablename.]columnname }
from tablename [corrname] {, tablename [corrname] }
[where qual]
[group by columnname {, columnname} [having qual] ]
[order by orderfieldname [sortorder]
    {, orderfieldname [sortorder] } ]
```

The behavior of select to a simple object is similar to that of a singleton query, but the syntax is different. For example:

```
select namefield = empname,
  workvariable = emp_hire_date
  from employee
  where empnum = :empnumfield
```

In this example, the employee record for the employee whose number is in *empnumfield* is retrieved from the Employee table. The column *empname* is placed directly on the form in *namefield*. The column *emp_hire_date* is placed in the variable *workvariable*.

Examples

Use *select to a complex object* to select information about an employee based on the value in the Empnum field; then use the commit statement to end the transaction:

```
deptform := select lname = last,
  fname = first
  from personnel
  where empnum = :empnum;
commit;
```

Use *select to a complex object* to select rows for all employees with an employee number greater than 100; then use the commit statement to end the transaction. The table name is held in a variable:

```
tbl := 'employee';
empform := select lname, fname,
  empnum = employee
  from :tbl
  where number >100;
commit;
```

Use *select to a complex object* to read into the table field Emptable all projects for the employee whose number is currently displayed in the Empnum simple field of the form; then use the **commit** statement to end the transaction:

```
emptable := select project, hours
  from projects
  where empnum = :empnum;
commit;
```

Use *select to a complex object* to select and sort employee names matching the qualifications of the *empdept* and *empsal* fields; then use the commit statement to end the transaction:

```
empform := select emplname = lname,
  empfname = fname
  from employee
  where qualification (dept = empdept,
    sal = empsal)
  order by emplname asc, empfname asc;
begin
  /* submenu code goes here */
end;
commit;
```

Use *select to a complex object* to read values from the Projects table into the Projform form, in which the simple fields and database columns correspond; then use the commit statement to end the transaction:

```
projform := select *
  from projects
  where empnum = :empnum;
commit;
```

Use a *select loop* with begin and end statement to perform an operation on the row matching the qualification criteria. Use the commit statement to end the transaction:

```
select * from emp
  where qualification (lname = lname)
begin
  /* statements, including queries, */
  /* which are executed once for every */
  /* row retrieved */
end
;
commit;
```

Use a *select loop* to a append records to an array:

```
/* name = char(...), salary = integer */
i = maxrow + 1;
select name, salary
  from emptable
begin
  emparray[i].name = name;
  emparray[i].salary = salary;
  i = i + 1 ;
  /* other statements here */
end;
```

Use a query to sort by the result (field) name, *person*, and not by the column name Ename.

```
newform := select person=ename, dept
  from employee
  order by person;
```

Use a singleton select to a record, selecting a row from the table Emp to the record *r_emp*:

```
/* r_emp = type of table employee */

r_emp = select * from employee
  where lname = 'Smith';
```

Select to an Array: Example

This example is a simple application which performs a search by reading a table into an array and searching the array. You enter a part name and the application returns the part number, if the part is found. The array is stored in memory. It is faster to search the array than the database table.

In the initialize section of the PartMenu, the only frame, the procedure SelectParts() is called. This selects the entire parts table into an array of type of table *part*, sorting it by *partname*.

When you choose Find, the only menu option, the application prompts for a part name, then calls the procedure GetPart(). GetPart() calls the procedure Binary_Search() to search the array. Binary_Search() returns the position of the array row containing the part in question, or 0 if that part was not found.

PartMenu then looks up the part number in the row returned by GetPart().

The application assumes that the data is not updated while you are searching for parts. The table is part of a read-only or rarely-updated database intended mainly for reference. If the data is likely to be updated by one user while another is doing a search, the application uses a time stamp. The time stamp must be stored in a database table and must be accessible to users updating and searching the Part table.

The source code for PartMenu and the procedures described above follow.

This is the 4GL code for PartMenu, the top frame of this application:

```
initialize = declare
  parttable = array of type of table part;
  name = char(16);
  pos = smallint;
begin
  callproc selectparts
    (parttable = parttable);
end
'Find' =
begin
  name = prompt 'Enter part name: ';
  pos = callproc getpart
    (parttable = parttable, name = name);
  if pos = 0 then
    message 'This part is not in the table'
      with style = popup;
  else
    message 'Part ' + :name + ': part # = ' +
      varchar(parttable[:pos].partno)
        with style = popup;
  endif;
end
'End', key frskey3 =
begin
  return;
end
```

This is the 4GL code for the procedure SelectParts(). This procedure selects the entire part table into an array, sorted by partname.

```
/* Input: parttable: the array to be selected into */
procedure selectparts ( parttable =
    array of type of table part ) =
begin
  parttable = select * from part
  order by partname;
end
```

This is the 4GL code for the procedure GetPart(). This procedure retrieves a particular part from an array of parts. You enter the name of the part to be found. GetPart() returns the position of the part within the array, or 0 if the part is not found.

```
procedure getpart ( parttable =
  array of type of table part,
  name = char(16)
    ) =
declare
  nrows = smallint;
  pos = smallint;
begin
  nrows = callproc arrayallrows( parttable);
  pos = callproc binary_search
    (arr = parttable, name = name, first = 1,
    last = nrows);
  return pos;
end
```

This is the 4GL code for the procedure Binary_Search(). This procedure performs a binary search on an array of parts. Binary_Search() returns the position of the part within the array, or 0 if the part is not found.

```
/*  Inputs: */
/*      arr - the array to be searched */
/*      name - the part name to be found */
/*      first, last - the beginning and end */
/*      of the array section being searched */
procedure binary_search ( arr =
  array of type of table part,
  name = char(16), first = smallint,
  last = smallint
    ) =
declare
  pos = smallint;
  middle = smallint;
begin
  if arr[first].partname = name then
    pos = first;
  elseif arr[last].partname = name then
    pos = last;
  elseif last - first <= 1 then
    pos = 0;    /* not found */
  else
    middle = first + (last-first)/2;
    if arr[middle].partname <= name then
      pos = callproc binary_search(arr = arr,
        name = name, first = middle,
          last = last);
    else
      pos = callproc binary_search(arr = arr,
        name = name, first = first,
          last = middle);
    endif;
  endif;
  return pos;
end
```

Sequence_value()

Returns a sequential value for a field.

Syntax

```
key = [callproc] sequence_value
  ( table_name = table_name , column_name =
  column_name [, increment =  increment, start_value
  = start_value])
```

key

Specifies an integer variable that holds the sequence value generated

table_name

Specifies a string expression whose value is the name of the table for which sequence_value() generates a key value. The table must already exist in the database.

column_name

Specifies a string expression whose value is the name of the column for which sequence_value() generates a key value. The column must already exist in the database.

increment

Specifies a positive integer that specifies the number of sequence values to be returned and added to the value stored in the table. Optional. Defaults to 1.

start_value

Specifies a positive integer that tells where to start generating sequence keys for an empty table, if sequence_value has never been called before for this table/column combination. Optional. Defaults to 1.

Description

The 4GL sequence_value() statement returns a positive integer that is automatically incremented whenever a value is fetched. You can use the sequence value as a surrogate key to assign to a row of a table or insert into other rows in other tables as a join attribute. This calling sequence operates like a 4GL procedure.

The sequence_value() procedure returns a positive integer that is a new sequence value (or the highest of a range of sequence values) for a unique input table and column.

The sequence_value() function generates sequential values by retrieving the next row from an extended system catalog called ii_sequence_values. This catalog contains a row for each unique table/column pair for which sequence values are defined.

The following table shows the attributes of ii_sequence_values:

sequence_owner

varchar(32). Owner of a table. Not nullable.

sequence_table

varchar(32). Name of a table. Not nullable.

sequence_column

varchar(32). Name of column in a table. Not nullable.

sequence_value

integer. Not nullable.

If a record exists in the catalog, the sequence_value() procedure increments it and returns the result. If not, the function increments the current maximum value of the column in the table.

If the table contains no rows which have non-null values for the column, it uses the *start_value* instead. If a record does not exist in the catalog, sequence_value() adds one. The default is 1. The return of a negative integer or 0 indicates an error.

Consider the following issues when using the sequence_ value() function:

- If you insert any rows to the table without using the sequence_value() function—for example, through QBF or the Terminal Monitor—these rows cannot have sequential key values, because only the sequence_value() function retrieves the next value from the ii_sequence_values catalog.

  To prevent non-unique values, the database table must have a structure of btree unique.
- There is no 3GL equivalent of this function; therefore, you cannot use a 3GL procedure to generate sequence values.

Transaction Handling with Sequence_value()

Your table can have gaps in the sequence of the values that sequence_value() generates. This is because the sequence value is returned before the insert is executed. Thus the ii_sequence_values catalog is updated even if the insert is not performed. If it is important that values be sequential with no gaps or duplicate values, place the sequence_value() and insert statements within the same transaction. Autocommit must be off.

However, it is not be efficient to do this when you have concurrent users and your insert statement takes a long time to execute. This is because the ii_sequence_values catalog is locked until the entire transaction is executed**.** If one user is adding new rows, other users trying to add rows at the same time must wait until the first transaction is completed.

Do not place the sequence_value() function and insert statement within the same transaction if you want to display the sequence value on the form. This is because the ii_sequence_values catalog remains locked until you actually save the new row to the database.

Sequence Values and System-Maintained Keys

Using the sequence_value() function lets you generate surrogate keys for rows that you add to a database table. You also can create unique keys by defining a column of the table to be a *system-maintained key* (described in the *SQL Reference Guide*).

The following table compares system-maintained keys and values generated by the sequence_value() function:

Attribute	**Sequence Values**	**System-Generated Keys**
Guaranteed Unique	Only if all values generated with sequence_value()	Yes (for either table or database as specified)
Guaranteed	Only if all values generated with sequence_value() and the call to	No (values generated

Attribute	Sequence Values	System-Generated Keys
Sequential	sequence_value() and the insert are done in the same transaction	at random)
Enforced by DBMS	No	Yes
Size	4 bytes (integer4)	8 bytes
Displayable on Form	Yes	No (appears as hexadecimal)
Can Be Copied to Another Database	Yes (values are maintained)	No (new values are assigned)

Example

The following statement shows the use of sequence_value() to obtain a surrogate key in the Employees table.

```
key = sequence_value
  (table_name = 'employees',
    column_name = 'employee_number' );
commit;
if (key > 0) then
  insert into employees
    (employee_number)
    values ( :key );
  commit;
```

When generating a large number of values, locking can occur**.** The following statement allows for generating 1000 numbers:

```
high = sequence_value (tablename = 'test',
        column_name = 'test_no',
        increment = 1000);
low = high - 999;
commit;
while low <= high do
  ... statements ...
low = low + 1;
endwhile;
```

Set_4gl

Sets the behavior of a tablefield, array, or record when a select statement returns zero rows.

Syntax

```
set_4gl (clear_on_no_rows = value)
```

The *value* is one of the following:

off | 0

Indicates the select target is left undisturbed if no rows are returned

on | 1

Indicates the select target is cleared if no rows are returned. This is the default behavior.

The *value* can also be a variable or a 4GL expression that evaluates to 0 or 1.

Description

This statement allows you to dynamically set the behavior of a tablefield, array, or record when selecting data**.** Use the clear_on_no_rows keyword to indicate whether the tablefield, array, or record is cleared if the select statement returns zero rows.

The clear_on_no_rows keyword has no effect if the select target is a form: the form is never automatically cleared.

You can check the current value of clear_on_no_rows with the inquire_4gl statement. See Inquire_4gl (see page 1029).

Examples

Use the variable *intvar* to change the behavior of the table field before issuing a select:

```
intvar = 0;
set_4gl (clear_on_no_rows = :intvar);
```

Change behavior back to clear table field behavior:

```
set_4gl (clear_on_no_rows = 1);
```

Or:

```
set_4gl (clear_on_no_rows = on);
```

Set_forms

Sets a variety of features of the FRS (FRS).

Syntax

```
set_forms objecttype {parentname}
  (set_forms_constant [(objectname)] = value
  {, set_forms_constant [(objectname)] = value})
```

objecttype

The type of object for which the value assignment sets features:

frs

Is FRS itself

form

Is a form

field

Is a simple field

row

Is a row in a table field

column

Is a column in a table field

menu

Is a menuitem on the form

The *objecttype* is a 4GL name and must be a character string.

parentname

The form, or form and tablefield, that contains the object being set. For a simple field or menu item, *parentname* is the form name. For a row or column in a table field, *parentname* is the form name and the table field name.

An empty string (' ') here indicates the current parent.

For example, in a set_forms field statement, *parentname* is the name of the form in which the field appears. The following statement sets the blink attribute on the field "emp_name" in the current form:

```
set_forms field ' ' (blink(emp_name)=1);
```

The *parentname* is a 4GL name.

parentname is not used when the object type for the statement is FRS or form.

set_forms_constant

FRS constant. Allows various features of the FRS to be set from within an application during program execution.

FRS constants are global; they apply to all forms in the application.

objectname

The object to which the *set_forms_constant* refers; a 4GL name. It can be a column, field, form, table-field row, menu, or variable of an Ingres data type (Default = current object).

Use a dereferencing colon (:) before the *objectname* if it is a local variable.

value

A character string or integer value as specified for each set_forms variant. The value must be within the limits specified for each set_forms constant.

Description

The 4GL set_forms statement allows you to set various features of the FRS (FRS) dynamically. The FRS handles the management of the forms that are displayed as the application runs.

Among the FRS features that can be set with this statement are:

- Display attributes for simple fields and table-field columns and rows
- Field validations and activations
- Mapping for control and function keys

Once set, the FRS options remain in effect until the application is exited or until a new statement clears them.

A corresponding inquire_forms statement exists for each variant of the set_forms statement. You can query and set features dynamically as required throughout an application.

CAUTION! The individual assignments specified in a set_forms statement are processed in reverse order**.** Thus, earlier assignments take precedence over later assignments in the same set_forms statement. For example, to specify both label and map for a key, specify label before map. Otherwise, the default label generated by the map overrides the label you specify.

The set forms statement has two variants, set_forms *formobject* and set_forms frs, determined by the *objecttype* being set. Each type is discussed in its own section below.

Set_forms Formobject (see page 1135), covers statements which take an *objecttype* of form, field, column, row or menu.

Set_forms FRS (see page 1142), covers only statements which take the *objecttype* frs.

Set_forms Formobject

In this discussion, set_forms *formobject* refers to the following variant of the set_forms statement:

```
set_forms objecttype (set_forms_constant)
```

The *objecttype* accepted in each case varies, but can be one of the following form components: column, field, form, menu, and row.

The set_forms_constant can be one of the following:

active

Enables or disables the specified menu item

change

Sets the change value of a form, field, or row and column

color

Sets color on terminals with color

(*display attributes*)

Turns the specified display attribute on or off.

The display variable refers to any one of the set_forms constants, such as blink or underline, that control display characteristics for the named *formobject*.

mode

Sets the mode for the current form

rename

Renames the specified menu item

The following subsections discuss each of these variants in alphabetical order according to the set_forms constant.

Set_forms Formobject (Active)

```
set_forms menu (active(menuitem)=value
   {, active(menuitem)=value})
```

Specifies whether the menu item is enabled or disabled. The *value* can be a keyword, integer constant, or integer variable as follows:

on | 1

Specifies that the menu item is enabled.

off | 0

Specifies that the menu item is disabled. When a menu item is disabled, it does not display in the window and cannot be selected

intvar

Specifies a 4GL expression evaluating to 0 or 1

When a menu item is disabled or enabled, the menu is redisplayed. Any menu key mappings that depend on the position of the items on the menu are adjusted accordingly. For example, suppose the function keys F1, F2, and F3 are mapped to the first three menu items as follows:

```
First(F1)   Second(F2)   Third(F3)   Fourth   End
```

Disabling the Second menu item results in F2 being mapped to the Third menu item and F3 being mapped to the Fourth menu item:

```
First(F1)    Third(F2)    Fourth(F3)    End
```

Enabling the Second menu item again results in the original mapping.

Set_forms Formobject (Change)

```
set_forms form (change = value)

set_forms field formname
    (change[(fieldname)] = value
    {, change[(fieldname)] = value})

set_forms row formname tablename [rownumber]
    (change[(columnname)] = value
    {, change[(columnname)] = value})
```

The three set_forms *formobject* statements, set_forms form (change), set_forms field (change), and set_forms row (change), reset the value of a change variable associated with a form, field, or a displayed column in a table field.

The *value* can be a keyword, integer constant, or integer variable as follows:

off | 0

Indicates that the data in a form, field, or table-field display has not been changed. Set the change variable to 0 if you want to clear the change variable to make it appear as though the runtime user had not entered data into the form object.

For a field, the change variable for is set to 0 automatically:

- At the start of a display loop
- When the field is cleared
- When a new value is placed into the field by the program.

For a form, the change variable is set to 0 only at the start of a display loop.

on | 1

Indicates that changes have been made to the data in a form, field, or table-field display

intvar

Specifies a 4GL expression evaluating to 0 or 1

The *rownumber* in the syntax for the set_forms row (change) statement is optional. It specifies a row number in the referenced table-field display, and can only reference rows that display actual data. If you do not specify a *rownumber*, the statement refers to the current row (currently occupied by the cursor). Do not specify the *rownumber* within an unloadtable loop. In this case, the statement always refers to the row just unloaded.

If a table field has an associated data set, then each value in the data set also has an associated change variable.

Set_forms Formobject (Color)

```
set_forms field formname
    (color [(fieldname)] = value
    {, color [(fieldname)] = value})

set_forms column formname tablename
    (color [(columnname)] = value
    {, color [(columnname)] = value})

set_forms row formname tablename [rownumber]
    (color[(columnname)] = value
    {, color[(columnname)] = value})
```

The set_forms field (color), set_forms column (color), and set_forms row (color) statements set the color display of individual fields, table-field columns, or rows, respectively, on color monitors.

The *value* of color can be an integer in the range from 0 to 7, or the name of an integer field with a value in that range. The default color code is 0, indicating a terminal's default foreground color.

The color represented by the value is determined by the terminal setting for each terminal or PC. The termcap entry contains color code that are associated with the color settings for a terminal or PC that supports color.

Set_forms Formobject (Display)

```
set_forms field formname
    (display [(fieldname)] =   value
    {, display [(fieldname)] = value})

set_forms column formname tablefieldname
    (display [(columnname)] = value
    {, display [(columnname)] = value})

set_forms row formname tablename [rownumber]
    (display[(columnname)] = value
    {, display[(columnname)] = value})
```

The set_forms field (*display*), set_forms column (*display*), and set_forms row (*display*) statements set display attributes for individual field, table-field columns, or rows dynamically.

The *display* parameter can be any of the FRS constants:

reverse

Turns reverse video on or off. The default is off.

blink

Turns blinking on or off. The default is off.

underline

Turns underline on or off. The default is off.

intensity

Changes the intensity of the display to half or bright intensity, depending on the monitor, if *value* is on. A *value* of off (the default) produces normal intensity.

displayonly

Turns the display only attribute on or off. The default is off.

Cannot be set for row.

invisible

Turns the invisibility attribute on or off. The default is off. Can be set in VIFRED.

Field only: Removes entire (table) field from view. Setting of field has higher precedence than setting of column for this attribute. (No column is visible unless the table is visible.)

Column only: Columns to right of invisible column shift to left, overlaying invisible column. If all columns of a table field are invisible, the entire table field is invisible. If a column setting changes, height of the table field row does not change.

Cannot be set for row.

normal

Setting to on specifies normal settings for all other display attributes except displayonly and invisible (that is, the other attributes are set to off). This applies to all settings except displayonly and invisible, which are not affected.

Note: Ingres sets this attribute to off if any other attributes are turned on; you cannot give it a *value* of off.

Cannot be set for row.

format

Changes the field or table-field format dynamically. The new format must be compatible with the field's existing data type and cannot exceed the original size (number of rows and columns). Data in the field must be valid. For allowable forms of the format *value,* see "display formats" in this guide*.*

Cannot be set for row.

inputmasking

Turns input mask behavior on or off. The default is off. Input masks are a format specification that can restrict what characters can be entered into each of the positions in a field. The *columnname* or *fieldname* is required for this parameter. Cannot be set for row. For more information on input masking, see the appropriate section of this guide*.*

The *value* can be a keyword, integer constant, or integer variable as follows:

on | 1

Turns the display attribute on

off | 0

Turns the display attribute off

intvar

Specifies a 4GL expression evaluating to 0 or 1

These parameters can also be set in VIFRED. The default values do not apply if you specified different settings by editing the form in VIFRED.

Set_forms Formobject (Mode)

```
set_forms form (mode = modetype)

set_forms field formname (mode(tablefieldname ) =
    modetype)
```

To set the display mode for the current form, use the 4GL set_forms form (mode) statement. Unlike the mode statement, set_forms form (mode) performs *no other* functions in the process of changing the mode. It *does not* reinitialize the form, break the display, or affect the data, with the exception noted in the next paragraph. Field and table field values are unaffected by the change.

To set the display mode for a table field, use the set_forms field *formname* (mode) statement.

CAUTION! Changing to or from Query mode for a table field clears the data from the table field.

You can inquire on the display mode of the current form using the complementary inquire_forms statement. See Inquire_forms (see page 1031) for information.

You can change the mode during submenu execution by using the set_forms form (mode) or set_forms field *formname* (mode) statements. Upon exiting the submenu, the form returns to the mode in effect before the submenu operation.

The *modetype* is a quoted character string constant, or the name of a field containing a character string constant, that specifies the form display mode. *Modetypes* for set_forms form (mode) and set_forms field *formname* (mode) are:

fill

Specifies that you can enter or change data (default mode)

update

Specifies that you can enter or change data

read

Specifies that you can examine but not change the data on the form

query

Specifies that you can enter comparison operators (such as =, >, <=) as well as data. This is required for a form on which you want to use the qualification function in a database query.

The following table shows the operations that take place when you dynamically set a table field mode:

Start Modetype	**New Modetype**			
	fill	**update**	**read**	**query**
fill	No action	Changes mode	Validates table field	Clears data set
update	Adds fake row if needed	No action	Validates table field	Clears data set
read	Adds fake row if needed	Changes mode	No action	Clears data set

- Validate table field. 4GL validates the field prior to changing the mode. If validation fails, the mode does not change.
- Clear data set. The table field, data set, and local variables are cleared. Hidden columns remain as previously defined. Moves the cursor to the first row/column cell of the table field. Performs any entry activations defined for the column if the cursor was not on the field before the mode change.
- Add fake row, if needed. A table field in either Update or Read mode contains no rows at all**.** Table fields in Fill mode always have at least one row.

Additional considerations in setting *modetype*:

- You cannot change the mode of an empty (uninitialized) table field.
- You cannot specify an unknown mode while the mode of the field in question remains unchanged.
- Aggregate derivation values can be invalidated (and cause a recalculation) when the mode of a table field changes. This can cause any derived fields dependent on the table field to be marked Invalid and recalculated.
- Setting the mode of the parent form has no impact on this feature.

Set_forms Formobject (Rename)

```
set_forms menu formname
  (rename(oldname)=newname
  {, rename(oldname)=newname})
```

Renames the menuitem specified by oldname to the string value, newname**.** Once a menuitem is renamed, it cannot be accessed by the old name. All references to the menuitem must be to the new name.

Set_forms FRS

The following sections discuss the variants of the set_forms statement where the *objecttype*, as discussed in the table Parameters for Set_forms, is always frs. The *set_forms_constant* can be any of those shown in the table below.

The following sections discuss these variants of the set_forms frs statement:

activate

Turns field activation mechanisms on or off

editor

Disables or enables system editor

getmessages

Enables or suppresses error messages on field value retrieval

label

Assigns an optional alias or identification label to a mapped key for display purposes

map

Sets mappings between control/function keys and FRS keys, FRS command, and menu items

mapfile

Specifies the application's mapping file

menumap

Turns on or off the display of key settings next to menu items

outofdatamessage

Sets the behavior for the "out of data" message

shell

Enables or disables the FRS shell command

timeout

Assigns a timeout period

validate

Turns validation mechanisms on or off

Each statement variant is discussed in its own subsection below, in alphabetical order by *set_forms_constant*.

Set_forms FRS (Activate)

The **set_forms frs activate** statement has the following syntax:

```
set_forms frs  (activate (activationtype) = value
  {, activate (activationtype) = value})
```

The 4GL set_forms frs (activate) statement sets default field activation types for the application as a whole. Field activation occurs when you perform an action (such as moving forward out of a field) that is specified by an activation type variable set to a value of on. You must use the before and after keywords when writing field activation code, even if the set_forms frs (activate) statement is used.

The *value* can be a keyword, integer constant, or integer variable as follows:

on | 1

Turns field activation on. This is the default.

off | 0

Turns field activation off.

intvar

Specifies a 4GL expression evaluating to 0 or 1

Activations take place only for fields for which 4GL code specifies the activation. The following table lists possible activation types. The default of before enables field activation before entry into a field; the other types perform activations after the cursor has entered a field.

Note: Be careful when writing entry applications, because an endless chain of entry activations can occur. For example, if fields "a" and "b" have entry activation code containing resume statements to each other, then an infinite loop results when you enter each field.

The possible activation types are:

before

Performs activation before the cursor enters the field or table field column from another field or table field column. Default activation type.

Enables or disables field entry activation, (activation when a specified field or table field cell is about to become the current item). Default is on.

keys

Performs activation when you press any function key mapped to an FRS key. The default is off.

menu

Performs activation when you press the Menu key. The default is off.

menuitem

Performs activation when you choose a menu operation or a key mapped to a menu operation. The default is off.

nextfield

Performs activation when the application user moves to the next field. The default is on.

previousfield

Performs activation when you move to the previous field. The default is off.

Set_forms FRS (Editor)

```
set_forms frs (editor = value )
```

The set_forms frs (editor) statement lets you disable the system editor to prevent access to the operating system from the system editor. The *value* can be a keyword, integer constant, or integer variable as follows:

off | 0

Disables the editor

on | 1

Enables the editor. This is the default, to allow for compatibility with earlier releases of 4GL.

intvar

Specifies a 4GL expression evaluating to 0 or 1

Set_forms FRS (Getmessages)

```
set_forms frs (getmessages = value )
```

The set_forms frs (getmessages) statement allows an application to suppress validation error messages that occur when an application attempts to retrieve a value from a field. The *value* can be a keyword, integer constant, or integer variable as follows:

off | 0

Disables getmessages

on | 1

Enables getmessages. This is the default.

intvar

Specifies a 4GL expression evaluating to 0 or 1

Set_forms FRS (Label)

```
set_forms frs
 (label(menuN|frskeyN|frscommand) = value
 { , label(menuN|frskeyN|frscommand) = value })
```

The set_forms frs (label) statement allows you to assign a designated character string label to an FRS key, menu item position, or FRS command (including arrow keys) for display purposes. When menumap is on, the label appears next to a menu operation in place of the default label for the key. This statement does not affect the key mapping itself.

The resulting *value* is a user-defined label that is a string constant or a program string variable, enclosed in quotes.

To map keys to menu operations, see the subsection below on set_forms frs (map).

If labels are assigned to *both* a menu operation *and* its associated FRS key, the FRS key label takes precedence. Any aliases created by the set_forms (label) statement remain in effect for the remainder of the application unless another label statement overrides it. For more information on function and control key mapping, see the appropriate sections of this guide.

Possible FRS objects for set_forms frs (label) are:

menu*N*

Specifies the *N*th menu item on the menu line. *N* must be in the range 1 to 25.

frskey*N*

Specifies an FRS key used in the application. *N* must be in the range 1 to 40.

frscommand

Specifies any of the FRS commands to which a control or function key can be mapped.

One of the *frscommand* options available is shell.

Windows: The *frscommand* spawns a command prompt from within the forms program in a manner similar to starting an editor on a field's contents. For security it is not active unless the application explicitly enables it.

UNIX: The *frscommand* spawns a Bourne shell (sh) (default) or c shell (csh) from within the forms program in a manner similar to starting an editor on a field's contents. For security it is not active unless the application explicitly enables it.

VMS: The *frscommand* spawns a DCL sub-process from within the forms program in a manner similar to starting an editor on a field's contents. For security it is not active unless the application explicitly enables it.

Set_forms FRS (Map)

```
set_forms frs
  (map(menuN|frskeyN|frscommand) =
    pfN |controlx | [up | down | right | left ]
    arrow )
      |[(label)]
  {, map(menuN|frskeyN|frscommand) =
     pfN|controlx| [up | down | right | left ] arrow )
      |[(label)]}

set_forms frs
  (map(menuN|frskeyN|frscommand) =
    pfN |controlx|[(label)]
```

(In the previous syntax, *x* represents any single uppercase or lowercase letter.)

The set_forms frs (map) statement allows mapping of menu operations, FRS keys and FRS commands to control and function keys (including arrow keys) at the application level. Key mapping at the application level takes precedence over installation and user mapping files. Each menu operation, FRS key, and FRS command can be mapped to only one control or function key.

You can also map keys in two other ways:

- Map a function/control key to a menu operation. Mapping a key to a menu operation causes a default *label* (PF3, PF4, etc.) to appear next to that menu operation when the menumap function is set to *on*. For information on changing this default label, see Set_forms FRS (Label).
- Map a function/control key to an FRS key associated with the same operation through program code.

Note: Because arrow keys on certain UNIX and VMS terminals send a control character (rather than an escape sequence), the implementation of your system must take this into account.

Mappings remain in effect for the remainder of the application unless overridden by another map statement. For more information on function and control key mapping, see the appropriate sections of this guide.

The following are the possible FRS objects and their values for set_forms frs (map):

menu*N*

Specifies the *N*th menu item on the menu line. *N* must be in the range 1 to 25.

frskey*N*

Specifies an FRS key used in the application. *N* must be in the range 1 to 40.

frscommand

Specifies any of the FRS commands to which a control or function key can be mapped

pf*N*

Specifies a function key. *N* must be in the range 1 to 40.

control*X*

Specifies a control key. *X* can be any single letter of the alphabet, or del (Delete key), or esc (Escape key).

Control cannot be abbreviated. On most terminals, Control-M is equivalent to the Return key, and Control-I is equivalent to the Tab key.

Control can be abbreviated ctrl. On most PCs Ctrl-M is equivalent to the Enter key, and Ctrl-I is equivalent to the Tab key.

label

Designates any alphanumeric string. It appears in place of the default label for a menu item. It also appears in the Keys operation of the Help facility.

One of the FRS commands available is shell. For characteristics, see Set_forms FRS (Label).

Set_forms FRS (Mapfile)

```
set_forms frs (mapfile = pathname | file_specification)
```

The 4GL set_forms frs (mapfile) statement allows an application's FRS key mapping file to be determined dynamically.

The value to assign to mapfile is the complete pathname or file specification for the mapping file, as appropriate for your operating system. It must be a character string constant enclosed in quotes or the name of a field containing a character string constant.

Further information on mapping files can be found in the appendixes of this guide.

Set_forms FRS (Menumap)

```
set_forms frs (menumap = value)
```

The 4GL set_forms frs (menumap) statement sets the display of the default menu mapping to keys. The *value* can be a keyword, integer constant, or integer variable as follows:

off | 0

Specifies that, when the display is off, or if no labels are defined, the key names do not appear.

on | 1

Specifies that, when the display is on, you see the key names in parentheses following the menu operations. This is the default.

intvar

Specifies a 4GL expression evaluating to 0 or 1

Default labels are set by the mapping files in the terminal definition. For details on mapping files, see the appendixes of this guide. Any labels assigned in set_forms frs (label) statements override the defaults.

Set_forms FRS (Outofdatamessage)

```
set_forms frs (outofdatamessage = value)
```

The set_forms frs (outofdatamessage) statement sets the behavior for the "out of data" message when a user attempts to scroll beyond the boundaries of a table field data set. The *value* can be a keyword, integer constant, or integer variable as follows:

off | 0

Indicates the message is not displayed

on | 1

Indicates the "out of data" message is displayed. This is the default.

bell | 2

Indicates that the forms system rings the monitor bell once, instead of displaying the "out of data" message

intvar

Specifies a 4GL expression evaluating to 0, 1 or 2

Set_forms FRS (Shell)

```
set_forms frs (shell = value)
```

The 4GL set_forms frs (shell) statement enables or disables the shell command key. The *value* can be a keyword, integer constant, or integer variable as follows:

off | 0

Disables the shell command key. This is the default.

on | 1

Enables the shell command key.

intvar

Specifies a 4GL expression evaluating to 0 or 1

For security, it is not active unless the application explicitly enables it.

Windows: The *frscommand* spawns a command prompt from within the forms program in a manner similar to starting an editor on a field's contents without any explicit coding.

UNIX: The shell option spawns a Bourne shell (sh) (default) or c shell (csh) from within the forms program in a manner similar to starting an editor on a field's contents without any explicit coding.

VMS: The shell option spawns a DCL sub-process from within the forms program in a manner similar to starting an editor on a field's contents without any explicit coding.

Set_forms FRS (Timeout)

```
set_forms frs (timeout = value )
```

The 4GL set_forms frs (timeout) statement specifies the number of seconds in the timeout period. The timeout value is normally set within the initialize block. For a timeout period, the *value* can range from 1 to 32767, that is, from 1 second to approximately 9 hours. A *value* of zero means no timeout is specified. Values greater than this range are truncated; negative values are set to zero and cause an error message.

If a timeout occurs, the *timeout block* is executed, defined by the keywords:

```
on timeout
  begin | {
    statements
  end | }
```

For further information and examples of timeout blocks, see the section Timeout Activations.

Set_forms FRS (Validate)

```
set_forms frs
  (validate (validationname) = value
  {, validate (validationname) = value})
```

The 4GL set_forms frs (validate) statement sets validations for the application as a whole, not for individual fields. Validation occurs when you perform an action (such as moving to the next field or column) that is specified by a *validationname* variable whose value is set to on. See Inquire_forms (see page 1031) for further discussion of this feature.

The following table lists the possible validation names for set_forms frs (validate). The *value* can be a keyword, integer constant, or integer variable as follows:

off | 0

Turns validation off

on | 1

Turns validation on for the named validation mechanism

intvar

Specifies a 4GL expression evaluating to 0 or 1

validationname can have any of the following values:

keys

Performs validation when you press any function key mapped to an FRS key. Default: off.

menu

Performs validation when you press the Menu key. Default: off.

menuitem

Performs validation when you choose a menu operation or a key mapped to a menu operation. Default: off.

nextfield

Performs validation when the application user moves to next field. Default: on.

previousfield

Performs validation when you move to the previous field. Default: off.

Examples

Clear the change variable for a field:

```
set_forms field empform
  (change(empname) = 0);
```

Set the change variable for two columns in row 2 of a table field:

```
set_forms row partsform partstable 2
  (change(partnumber) = 1,
  change(partdescription) = 1);
```

Cause the current field in the current form to appear in reverse video and blink:

```
set_forms field '' (reverse = 1,
  blink = 1);
```

Turn off blinking in the *partname* column in the *partstbl* table field of the current form:

```
set_forms column '' partstbl
  (blink (partname) = 0);
```

Change the mode of the current form to the mode designated in the *modefield* field of the form:

```
set_forms form (mode = modefield);
```

Cause a field activation to occur whenever you backtab or choose the Menu key or an FRS key:

```
set_forms frs (activate(previousfield) = 1,
  activate(menu) = 1, activate(keys) = 1);
```

Map the Control-H key to menu item 3, and equate the label "CTRL-H" with the same operation. When menumap is on, the label "CTRL-H" appears next to menu item 3, and the associated operation is invoked when you press the Control key and the "h" key:

```
set_forms frs (map(menu3) = 'controlh',
  label(menu3) = 'CTRL-H');
```

Map the uparrow key to menu item 4, and equate the label "SCROLLUP" with the same operation. When menumap is on, the label "SCROLLUP" appears next to menu item 4, and the associated operation is invoked when you press the Scrollup key:

```
set_forms frs (map(menu4) = 'uparrow',
  label(menu4) = 'SCROLLUP');
```

Equate Function key 3 with FRS key 5, so that any key activation operation utilizing FRS key 5 is activated when you press Function key 3:

```
set_forms frs (map(frskey5) = 'pf3');
```

Turn on the menu map display and set validation on the Menu key:

```
set_forms frs (menumap = 1,
  validate (menu) = 1);
```

Perform a validation for the current field whenever you press any function key mapped to an FRS key.

```
set_forms frs (validate (keys) = 1);
```

Turn on all possible field and column validations for every frame in the application:

```
set_forms frs (validate(nextfield) = 1,
  validate(previousfield) = 1,
  validate(menu) = 1,
  validate(keys) = 1,
  validate(menuitem) = 1);
```

Set_sql

Sets options for a DBMS session.

Syntax

```
set_sql (set_sql_constant = value
     {, set_sql_constant = value})
```

set_sql_constant

Specifies one of the following DBMS constants:

connection_name

Specifies the name of the session. Can be a 4GL name. A varchar of length 128.

errortype

Specifies the type of error number returned to errorno; options are:

- Genericerror generic error number (default)
- dbmserror local DBMS error number

programquit

Specifies whether Ingres aborts on the following errors:

- An application that is not connected to a database issues a query
- The Ingres DBMS fails
- Communications services fail

Set programquit to 1 to abort on these errors.

session

Specifies an integer identifier for the session. Can be a 4GL name.

value

Specifies a character string expression (for connection_name or errortype) or integer expression (for programquit or session)

Description

The 4GL set_sql statement lets you set dynamically any of the following options for a DBMS session:

- The type of error number returned to errorno in an inquire_sql statement
- Whether Ingres aborts on certain errors
- The name and identifier for a session connection

Example

Specify that local DBMS error numbers are to be returned by an inquire_sql statement on errorno and sqlcode:

```
set_sql (errortype = 'dbmserror');
```

Sleep

Suspends operation of the application for a specified number of seconds.

Syntax

```
sleep integervalue
```

integervalue

Specifies the duration of the specified pause in an application, in number of seconds. The number must be an integer literal or variable.

Description

The 4GL sleep statement stops the application for the number of seconds you specify.

When used with the message statement, it allows a menu-line style message to remain on the frame long enough for an application user to read it. You cannot need a sleep statement if the statements following message requires some time to process.

Example

Display a menu-line style message for three seconds:

```
message
  'Please enter an employee number: ';
sleep 3;
```

Unloadtable

Unloads the contents of the data set underlying a table field or array.

Syntax

```
unloadtable tablefieldname
  [(fieldname = columnname
  {, fieldname = columnname})]
begin | {
  statement {; statement}
end | }

unloadtable arrayname
  [(fieldname = attributename
  {, fieldname = attributename})]
begin | {
  statement {; statement}
end | }
```

tablefieldname

Specifies the name of the table field whose data you are unloading

arrayname

Specifies the name of the array whose data you are unloading

fieldname

Specifies the name of a form field, variable, or record attribute into which you are unloading a table-field column or FRS constant

columnname **or** ***attributename***

Specifies the name of the table-field column or array attribute being unloaded. A 4GL name.

You can specify the special FRS attributes _state or _record as column or attribute names.

See Forms-Control Statements (see page 852) for more information about values of _state and _record.

Description

The 4GL unloadtable statement lets you scan the data set of an array or table field, one record at a time. Unloadtable causes a loop to execute once for each record in the data set of an array or table field, whether or not the record is currently displayed on the form. As each record is unloaded, all the statements associated with the unloadtable statement are performed. At this point, the next record in the data set is unloaded, and the statements are performed for that record.

The loop continues until either all the records in the data set (including records whose state is Deleted) are processed or an endloop statement is encountered. A validation error also terminates the unloadtable loop.

The statement assigns values from a table-field's columns or from the attributes of an array record, or row status from the FRS constants, _state and _record, to specified variables.

The unloadtable statement operates on arrays in a similar manner. However, because arrays do not define 'display' rows, the statement uses the value of the _record constant as an implicit reference to the index number of the array record.

The unloadtable statement also provides an implicit definition for the _state constant for the records in the array. The possible values for _state are the same as those supported for table fields, except that a state of Undefined (0) is not allowed for arrays. The values for _state in an array are completely under the control of the 4GL program code (because the end users cannot "fill" or modify an array through the FRS).

The unloadtable statement processes records in the following order:

- Non-deleted records are processed first.

  Within non-deleted records, records are processed in ascending order by record number (the value of _record).
- Then deleted records (that is, those whose state is Deleted) are processed.

  Within deleted records, the most recently deleted record is processed first, then the next most recently deleted record, and so on.

The unloadtable statement has the following restrictions:

- An unloadtable loop cannot include deleterow or insertrow statements against the table field or array being unloaded.
- An unloadtable loop cannot include a database retrieval to the table field being unloaded, because that clears the table field. However, you can select into a column of the table field being unloaded.

- An unloadtable loop unloading an array cannot invoke any of the following array functions: arrayclear, arrayinsertrow, arrayremoverow, or arraysetrowdeleted.
- You cannot nest unloadtable statements. For example, if *array2* is an attribute of *array1*, you cannot use an unloadtable statement with *array2*.

The entire unloadtable statement, including the statement list, is considered a single 4GL statement. Use these statement conventions:

- Enclose the statement list either with braces or the keywords begin and end.
- Place the statement separator (; or ,) at the end following the closing brace or end keyword if the statement is followed by other statements.
- Do not use a statement separator before the brace (or begin) that opens the statement list.

Within the unloadtable loop, certain expressions have special meanings:

- If you are unloading the table field *tablefield*, then within the unloadtable loop, expressions of the form *tablefield*.*column* refer to the column in the currently unloaded row (rather than the column in the row the cursor occupies). To access a column in the row the cursor occupies, use an explicit row number.
- If you are unloading *array*, then within the unloadtable loop, expressions of the form *array*.*attribute* refer to the attribute in the currently unloaded record. (The compiler does not allow this type of expression in any other context.)

For information on detecting user changes to the fields in the data set, see the section Inquire_forms Row in Inquire_forms Field (see page 1035). For additional information on the unloadtable statement, _state, and _record, see Forms-Control Statements (see page 852).

Copying Table Fields and Arrays

To copy an array to a table field or a table field to an array, use direct unloading and insertion. This constitutes a copy rather than an assignment, as shown in the following examples:

```
i = integer not null;
x = array of type of table field tf ;

'Copy_tf_to_array '=
begin
    i = ArrayClear(x) ;
    i = 0;
    unloadtable tf
    begin
      if (tf._state = 4) then
        insertrow x[i] (col1 = tf.col1, ...);

        /* New row is x[i + 1], not x[i]. */
        /* _state = 4 may not be specified */
        /* on an insertrow. */

        deleterow x[i + 1];
      elseif (tf._state <> 0) then
        insertrow x[i] (col1 = tf.col1, ...,
          _state = tf._state);


        /* New row is x[i + 1], not x[i]. */
        /* _state = 0 may not be specified */
        /* on an insertrow into an array. */

        i = i + 1;
      endif;
    end;
end
```

```
i = integer not null;
r = integer not null;
x = array of type of table field tf;

'Copy_array_to_tf '=
begin
    clear field tf;
    inquire_forms table tf_form
      (r = maxrow(tf));
    i = 0;
    unloadtable x
    begin
      if (x._state = 4) then
        insertrow tf[0] (col1 = x.col1, ...);

        /* New row is tf[1], not tf[0]. */
        /* _state = 4 may not be specified */
        /* on an insertrow */

        deleterow tf[i] ;
      else
        insertrow tf[i] (col1 = x.col1, ...,
          _state = x._state);

        /* New row is tf[i + 1], not tf[i],*/
        /* except that when i = r, */
        /* the table field is scrolled */
        /* up 1 row and the new row is tf[r]. */


        if i < r then
          i = i + 1;
        endif;
      endif;
    end;
    scroll tf to 1;
end
```

Using Unloadtable in Performing Calculations

The derived field functionality usually represents the best alternative when you want to total the values in a column of a table field. However, the unloadtable statement offers an alternative method, as in the following example:

```
'SumSalaries' =
begin
  tot := 0;
  unloadtable emptable
  begin
    tot := tot + emptable.salary;
  end;
  total := tot;
end;
```

When the application user chooses the SumSalaries menu operation, the unloadtable statement unloads each row of the table field *emptable* and adds the *salary* column to a running total in the hidden field *tot*. After unloading all the values, the running total is assigned to the field *total*.

For information about using derived fields for calculations, see Derived Fields and Columns (see page 814).

Examples

Use unloadtable with pattern matching to scroll the table field to the first row whose data starts with the characters you specify. If you enter "Smith" the pattern is set to 'Smith%' and retrieves the first row beginning with the name Smith, Smithson, Smithburg, etc. The display then scrolls to the selected row.

```
pattern := name + '%';
unloadtable emp (row = _record)
begin
  if emp.name like :pattern then
    scroll emp to row;
    endloop;
  endif;
end;
```

The example below updates a mailing list on the basis of the _state of each data set row. The hidden table-field column called *maillist.old* is the key to the updating, because it contains the original value of the Name field when the row was originally loaded from the database into the data set:

```
'Writelist' =
begin
  unloadtable maillist (rowstat = _state)
  begin
    if rowstat = 1 then
      /* Add row to database */
      repeated insert into mailtable
        (name, address, city, state)
        values (maillist.name,
          maillist.address,
          maillist.city,
          maillist.state);
    elseif rowstat = 3 then
      /* Update the row in the database */
      repeated update mailtable
        set name = :maillist.name,
        address = :maillist.address,
        city = :maillist.city,
        state = :maillist.state
        where name = :maillist.old;

    elseif rowstat = 4 then
      /* Delete the row from the */
      /* database */
      repeated delete from mailtable
        where name = :maillist.old;
    endif;
  end;
  commit;
end
```

Update

Updates the values of the columns in a database table.

Syntax

```
[repeated] update [owner.]tablename [corrname]
  [from fromsource {, fromsource}]
  set columnname = expression {, columnname =
    expression}
  [where qual]
```

[*owner.*]***tablename***

Specifies the name of the database table to be updated and, optionally, the owner of the table. The table name can be a 4GL name; however, if a 4GL variable is used, an owner cannot be specified, unless the owner name is included as part of the value. In other words, the variable can contain the value *owner.tablename.*

corrname

Specifies the correlation name for the table. This can be used in *qual* or any of the expressions to identify which table a column is to be taken from (for example, *corrname.columnname)*. If *corrname* is omitted, *tablename.columnname* can be written.

fromsource

The name of the source of the columns that appear in *qual* or any of the expressions. A source can be a table or an outer join between tables. A table is specified with the syntax:

```
[owner.]tablename [corrname]
```

A join between tables is specified with the syntax:

```
source join_type join source on
  search_condition
```

where a *source* can be a table name or a join. For more information about the from clause, see Select (see page 1094).

columnname

Specifies the name of a particular column to be updated in the table. A 4GL name.

expression

Specifies a value to which a column is to be set by the update statement.

Expression is typically a constant, a field, a table-field column, or a database expression. *Expression* can refer to database table columns. *Expression* can also be built from a function or an arithmetic operation using constants or single-valued fields. In addition, it can be built from a global or local variable, a literal, a record or an array record attribute.

qual

Specifies a logical expression that restricts the query to certain conditions. It cannot include 4GL names. Use the qualification function.

Description

The 4GL update statement modifies the values of columns in a database table with values from the current form or variables. The values can originate from simple fields, a table field, the columns of a table field, record attributes, or attributes of an array record. The syntax for the update statement in 4GL is identical to the SQL version, except that the set clause is modified to accommodate the assigning of values from the form or variables to columns in a database table.

The 4GL update statement updates the columns listed in the set clause for all rows in the table that satisfy the search condition in the where clause.

- The set clause contains column values from the table being updated or any table or tables listed in the from clause.
- The where clause contains a qualification function.
- The where clause can be stored in a variable. However, in this case, you cannot use the qualification function.

Because the update statement can operate on several rows or records at a time, formulate the where clause carefully to avoid unintentional updates.

If you expect to use a query frequently, overall performance improves if you use the repeated clause.

Precede each field name in the set clause with a colon to dereference it and eliminate possible conflicts between the field name or table-field column and the name of a database column. Use a colon to dereference fields in the where clause.

You can use the all option as a suffix with either a form name or a table-field name. The table must have a column that corresponds to each of the displayed fields or table-field columns on the form. When using all:

- Attach the reserved word all to the name of the form or table field, not to the name of the database table.
- In the case of the table field, use all to indicate all the columns of the current row or of the row specified by *integerexpr*, if there is one.
- The asterisk (*) substitutes for the list of database columns.

Updating a database table with values from many rows in a table field generally involves using the update statement in conjunction with the unloadtable statement. The update statement is located inside an unloadtable loop. See the section Unloadtable for additional information.

Examples

Update rows in the Projects table with values from the current form:

```
repeated update projects
  set hours = :hours, duedate = :enddate
  where name = :name;
commit;
```

Update the Personnel table with a computed value:

```
update personnel
  set sal = :salary * 1.1
  where empno = :empno;
commit;
```

Update row in the Personnel table with values from all the simple fields in form called Deptform:

```
update personnel
  set * = deptform.all
  where idnum = :idnum;
commit;
```

Update the salaries of all employees in job category acc, using the value for a standard raise in the table Dept:

```
update employee e
  from dept d
  set salary = d.std_raise * e.salary
  where e.jobcat = 'acc'
    and d.dname = e.dname;
```

Update the Employee table to change Address using the value in the *newaddr* attribute of the record *emprec*:

```
update employee
  set address = :emprec.newaddr
  where name = :emprec.name;
```

Update the Employee table with changes to employee titles using the value in the *title* attribute of array *emparray*:

```
update employee
  set emptitle = :emparray.title
  where emptitle = :emparray.old_title;
```

Validate

Performs the field validation checks previously established in the Visual-Forms-Editor (VIFRED).

Syntax

```
validate

validate field fieldname
```

fieldname

Specifies the name of a simple field or table field. A 4GL name.

Description

The 4GL validate statement checks the contents of the form against the validation criteria you establish for it using the ABF FormEdit operation to access VIFRED. If any field fails a check, an error message is displayed, the application breaks out of the operation currently in progress, and the cursor moves to the field that failed the check.

The validate statement initiates a validation check immediately upon execution, independent of any action taken by you. This differs from validations that occur as a result of the set_forms frs (validate) statement, in which the validation takes place when you perform an action (such as moving to the next or previous field).

The two variants of the validate statement are validate and validate field:

- Validate has no parameters. Validation checks are performed on every field in the current form for which validation checks are specified in VIFRED. At minimum, basic data type checking is done.
- Validate field has the single parameter, fieldname. This statement performs checks on the specified field or table field only. If you specify a table field, the data in all the table-field's displayed rows is validated.

To validate single rows in a table field, use the validrow statement, discussed in the following section.

When the form is in Query mode, the validate statement validates a field's datatype and query operators.

Examples

Validate the data in the field called Salary:

```
validate field salary;
```

Validate the data in the field on the form with the name found in the field Valfield:

```
validate field :valfield;
```

If the field Valfield contains the word "Salary", the above example validates the data in the Salary field.

Validrow

Performs a validation check on the columns in a specified row in a table field.

Syntax

```
validrow tablefieldname[[integerexpr]]
  [(columnname {, columnname})]
```

tablefieldname

Specifies the name of the table field subject to validation. This is a 4GL name (unless you have included *integerexpr* and its enclosing brackets).

integerexpr

Specifies the number of the row in the table field to be validated

columnname

Specifies the name of the column to be validated in the specified row. This is a 4GL name.

Description

The 4GL validrow statement performs a validation check specified for a table-field row on the current form. Validations are specified through the ABF FormEdit operation to access VIFRED. If a column fails a check, an error message is displayed.

The validrow statement performs a validation check immediately upon execution, independent of any action taken by you. This differentiates validrow checks from those set by the set_forms frs (validate) statement, which take place when you perform an action (such as moving to the next or previous field).

The validrow statement validates the row on which the cursor is currently resting unless an integer expression is specified in brackets. In this case, it validates the row with that number. Make sure that the cursor is on the table field if you execute validrow without specifying an integer expression.

If a list of columns is specified in the validrow statement, only the values in those columns are validated. Omitting the column list causes all columns in the row to be validated.

Validrow is disallowed for arrays, because arrays do not have validations.

Examples

Validate the current row in the Partstbl table field:

```
validrow partstbl;
```

Validate the columns called Partno and Cost in the third row in the table field called Partstbl:

```
validrow partstbl[3] (partno, cost) ;
```

While

Repeats a series of statements while a specified condition is true.

Syntax

```
[label:] while condition do
      statement; {statement;}
    endwhile
```

label

Specifies a character string identifying each while statement. Allows an endloop statement to break out of a nested series of while statements to a specified level. The *label* precedes the keyword while, and is followed by a colon. Must be a unique alphanumeric identifier, and not a field name or keyword.

condition

Specifies a logical (Boolean) expression evaluating to True or False

Description

The 4GL while statement executes the series of statements between the keywords do and endwhile as long as the condition represented by the logical (Boolean) expression remains true. The condition is tested only at the start of each loop; if values change during execution of the loop so that the condition becomes false, execution continues through the current iteration of the statement list, unless an endloop statement is encountered.

Breaking Out of While Loops

Use the endloop statement to break out of a while loop. Endloop immediately closes the loop, and execution continues with the first statement following endwhile. For example:

```
while condition1 do
  statementlist1
  if condition2 then
    endloop;
  endif;
  statementlist2
endwhile;
```

If *condition2* is true, *statementlist2* is not executed in that pass through the loop, and the entire loop is closed.

You can use nested while statements in 4GL, and you can control breaking out of nested loops by using labels, which allow the endloop statement to break to a specific level.

For example:

```
label1: while condition1 do
  statementlist1
  label2: while condition2 do
    statementlist2
    if condition3 then
      endloop label1;
    elseif condition4 then
      endloop label2;
    endif;
    statementlist3
  endwhile;
  statementlist4
endwhile;
```

There are two possible breaks out of the inner loop. If *condition3* is true, endloop closes both loops, and control resumes at the statement following the outer loop. If *condition4* is true, only the inner loop closes, and execution continues at the beginning of *statementlist4*.

If you do not specify a label after endloop, only the innermost loop currently active is closed. See the section on endloop in this chapter for additional information.

Examples

Repeat a prompt five times or until a valid answer is given:

```
answer := ' ';
requests := 0;
while (lowercase(left(answer,1)) != 'y' and
  lowercase(left(answer,1)) != 'n' and
  requests < 5) do
  answer := prompt 'Please answer Y or N:';
  requests := requests + 1;
endwhile;
```

Use the while statement to handle deadlock:

```
succeed := 0;   /* Set to 1 when and if */
                /* insert is successful */
tries := 1;
ingerr := 0;

A:  while succeed = 0 and tries <= 10 do
    insert statement to try to insert
    into database

        inquire_sql (ingerr = dbmserror);
        if ingerr = 0 then

      /* Insert completed successfully -  */
      /* commit the change           */

      commit;
      succeed := 1;
    elseif ingerr=4700 then
      /* Error due to deadlock - try again */
      tries := tries + 1;
    else
      endloop A;
    endif;
  endwhile;
if succeed = 1 then
  message 'Insert completed successfully';
  sleep 3;
else
  message 'Insert failed';
  sleep 3;
endif;
```

WriteFile( )

Writes a file.

Syntax

```
[returnfield = ] [callproc] writefile( handle =
  handle
  , item {, item } )
```

returnfield

Specifies the name of field to which the result is returned. Integer.

handle

Specifies the file identifier used by the 4GL file access functions. Integer.

item

Specifies the data that you are writing to the file. If *item* is a variable, 4GL writes as many bytes as are in the item's data type.

Description

The writefile() function writes data to a file that was previously opened with the openfile() function. The file must be open and the file's handle must be known to 4GL.

For files of type "text," you can only write string data types. Trailing blanks are removed from text and varchar fields. Null values are not accepted. You must convert null values with the ifnull function. For example:

```
writefile (handle = handle,
    ifnull(:desc, 'null indicator text'));
```

For a binary type file, the total size of the items written to the file must be the same as the file's record size. You can declare the record size in the openfile() statement, otherwise 4GL calculates it from the item list size indicated with the first read or write to the file.

The procedure returns 0 if the function completes without error.

Example

In this example, each row of the description column in the table tbl is unloaded into the variable *one_row*. Each row is then written to the file with the handle *file_no*.

```
unloadtable tbl (one_row = description)
begin
  if (tbl._state = 4) then
    endloop;
  endif;
  status = callproc writefile(handle = file_no, one_row);
  if status != 0 then
    endloop;
  endif;
```

end;

Chapter 22: Using 3GL Procedures

This section contains the following topics:

The statements documented in this chapter allow you to access structured data in your 4GL application, such as records and arrays, from a 3GL procedure. All 3GL routines using these statements can be used only in 4GL applications.

Passing Records and Arrays Between 4GL and 3GL Routines

A common reason for passing data to 3GL is enhanced performance when performing complicated mathematical calculations. If you are passing a small number of scalar data items, you can use a 3GL procedure. However, if the amount of scalar data you pass exceeds the number of parameters allowed on your system, or you want to pass structured data, such as an array or a record, you access the data from an embedded SQL procedure.

When accessing structured data in 4GL from 3GL, use a special set of embedded SQL statements that have the following syntax:

```
[margin] exec 4gl 4gl_statement [terminator]
```

Note that this syntax is similar to that for embedded SQL and SQL/Forms.

These statements are documented in this chapter. All 3GL routines using these statements can be used only in 4GL applications.

Use an embedded SQL procedure in conjunction with 4GL to perform the following functions:

- Getting and putting record and array attributes
- Dynamically accessing record and array attributes by a describe operation using the SQLDA
- Indexing into arrays to get or put objects
- Inserting rows into, deleting rows from, and removing rows from arrays
- Clearing arrays
- Accessing global variables and constants

Passing Structured Data

Structured data is passed to 3GL as a *handle*. Reference the handle in the exec 4gl statements whenever you want to indicate the record or array.

The handle is passed as a 4-byte integer. To store a handle passed in from 4GL, use the long data type on most platforms.

Handles passed to a 3GL routine are not valid after a return to 4GL. For this reason, the 3GL routine does not store a handle in a static variable. After control is returned to 4GL, a later call from 4GL to 3GL cannot use the handle stored earlier in the static variable.

Checking for Errors

Use the inquire_4gl statement to check for errors resulting from the exec 4gl statements. By default, any such errors are displayed on the screen. To turn off the display of these errors, use the set_4gl statement with the messages keyword.

Neither the global error handler that set_sql defines nor the sql whenever statement apply to exec 4gl errors. An error handler specified with the iiseterr function is not called when an error occurs.

Summary of Exec 4GL Commands

The following functions, and their associated statements, are available in 3GL when you pass objects and arrays from 4GL. You must always place the exec 4gl keywords before each statement.

Function	**Statement Name**
Access global variables and constants	get | set global variable get global constant
Call 4GL from 3GL	callproc
Clear an array	clear array
Get attribute(s) of a record	get attribute
Get record or array information. Get error status/text from previous operation	inquire_4gl
Get list of attributes of class	describe *object* into *descriptor*
Insert row into an array	insertrow
Remove row from an array	removerow
Set attribute(s) of a record or array	set attribute
Set error display status	set_4gl
Set row in an array as deleted	setrow deleted

Callframe, Callproc

Call 4GL.

Syntax

```
exec 4gl callframe framename ({ param =
  [byref(]variable[:ind][)] }) [into :variable:ind]

exec 4gl callproc procname ({ [param =]
  [byref(]variable[:ind][)] }) [into :variable:ind]
```

The parentheses enclosing the parameter list are optional if there are no parameters.

procname

Specifies the name of the procedure to call. String.

framename

Specifies the name of the frame to call. String.

param

Specifies the name of the parameter. The parameter must be in the called frame or procedure's parameter list.

variable

Specifies a variable in the calling 3GL procedure

ind

Specifies a null indicator variable

Description

This allows a 3GL procedure, which is part of a 4GL application, to call back into 4GL, using almost exact 4GL syntax. You can call:

- 4GL frames
- 4GL procedures

  The parameter name is required for 4GL procedures.

- 3GL procedures

  3GL procedures must be registered as part of the 4GL application. The parameter name is illegal for 3GL procedures.
- Database procedures

  Database procedures must be registered as part of the 4GL application. The parameter name is required for DBMS procedures.

You cannot pass queries to called components.

These statements are similar to the corresponding 4GL statements:

callproc procedure( );

callframe frame;

Clear Array

Clears an array.

Syntax

```
exec 4gl clear array arr
```

arr

Specifies the handle of the array to clear. Integer.

Description

The clear array statement removes all of the rows from the specified array. This is similar to the 4GL ArrayClear( ) procedure.

Describe

Get list of attributes of class.

Syntax

```
exec 4gl describe object into descriptor;
```

object

Specifies the handle of the record or array to be described. Integer.

descriptor

Specifies the name of the descriptor for the SQLDA

Description

The describe statement creates descriptors for the record's or array's attributes in the SQLDA structure specified by *descriptor*. For an array, it creates descriptors for the row's type.

This statement is similar to the FRS describe statement.

Get Attribute

Get an attribute from an array or record.

Syntax

```
exec 4gl get attribute record | array index
  (:var[:ind] = attribute [,:var[:ind] = attribute]);

exec 4gl get attribute record | array index
  using descriptor;
```

record

Specifies the handle of the record. Integer.

array

Specifies the handle of the array. Integer.

var

Specifies the 3GL variable to receive results

ind

Specifies a null indicator variable

attribute

Specifies the name of the attribute

index

Specifies the array index

descriptor

Specifies the name of the SQLDA descriptor

Description

The get attribute statement copies the attributes of a record or array row into 3GL variables. The named attributes must appear in the record or array, and the types of the attributes must be compatible with the types of the target variables.

The using *descriptor* form gets the attributes indicated in the SQLDA set up by the describe statement. See Describe (see page 1180).

Get Global, Set Global

Access global variables and constants

Syntax

```
exec 4gl get global variable (:var[:ind] = name)

exec 4gl set global variable (name = :var[:ind])

exec 4gl get global constant (:var[:ind] = name)
```

var

Specifies a 3GL variable

ind

Specifies a 3GL Null indicator variable

name

Specifies the name of the global variable or global constant being retrieved

Description

This allows 3GL to access 4GL global variables and constants, including structured data, which requires *var* as a handle.

Inquire_4gl

Get record or array information

Syntax

```
exec 4gl inquire_4gl(variable[:ind] = 4gl_constant
  [(array)])
```

variable

Specifies a 3GL variable

array

Specifies the handle of the array

4gl_constant

Specifies a constant containing the information you are seeking from 4GL about the record or array.

The following table of 4GL_constants describes the values that can be used in place of *4gl_constant* in the syntax.

Name	Type	Meaning
allrows	integer	The total number of rows in the specified array, including deleted rows This is similar to the 4GL procedure ArrayAllRows( ).
firstrow	integer	The index of the first (lowest-numbered) deleted row This is similar to the 4GL procedure ArrayFirstRow( ).
lastrow	integer	The number of non-deleted rows in the specified array This is similar to the 4GL procedure ArrayLastRow( ).
IsArray	integer	Returns 1 if array, 0 if not
RecordTypename	character	For a record, the name of the record type. For an array, this is the name of the row type.
errorno	integer	The number of the error for the previous operation, 0 if OK
errortext	character	The text associated with the error for

Name	Type	Meaning
		the previous operation

ind

Specifies a null indicator variable

Description

The inquire_4gl statement gets information about a record or array. The inquire_4gl statement can return:

- The name of the object's record type (recordtype name)
- Whether the named structure is an array
- Error status from the previous operation
- Error text from the previous operation
- Information about the number of rows in the array

InsertRow

Insert into array.

Syntax

```
exec 4gl insertrow array (rownumber = integer);
```

array

Specifies the handle of the array

integer

Specifies the number of the row to be added

Description

The insertrow statement adds a new row, containing default data, to the array.

This is similar to the 4GL procedure ArrayInsertRow( ).

Removerow

Remove from array.

Syntax

```
exec 4gl removerow array (rownumber = integer);
```

array

Specifies the handle of the array

integer

Specifies the row number of the record to be removed

Description

The removerow statement removes a row completely from an array.

This is similar to the 4GL procedure ArrayRemoveRow( ).

Set Attribute

Set attribute or attributes of a record or array.

Syntax

```
exec 4gl set attribute record | array index
  (attribute = :var[:ind] [(attribute = :var[:ind]])]

exec 4gl set attribute record | array index
  using descriptor
```

record

Specifies the handle of the record

array

Specifies the handle of the array

index

Specifies the index number of the variable; indicates the row of the array variable which is to be set

attribute

Specifies the name of the attribute

var

Specifies a 3GL variable

ind

Specifies a 3GL Null indicator

descriptor

Specifies the name of the SQLDA descriptor

Description

The set attribute statement copies the values from 3GL variables into the attributes of a record or array row. The named attributes must appear in the record or array, and the types of the attributes must be compatible with the types of the variables.

The using *descriptor* form sets the attributes indicated in the SQLDA set up by the describe statement. See Describe (see page 1180) for more information.

Set_4gl

Set error display status

Syntax

```
exec 4gl set_4gl (messages = value)
```

value

Specifies 1 to display error messages from exec 4gl statements, 0 to suppress them. Integer.

Description

The set_4gl statement turns error reporting to the screen on and off. By default, errors from exec 4gl statements are displayed.

This is similar to the set_frs getmessages call.

Setrow Deleted

Set deleted in array.

Syntax

```
exec 4gl setrow deleted array (rownumber = integer)
```

array

Specifies the handle of the array

integer

Specifies the number of the row to be deleted

Description

The setrow deleted statement makes a non-deleted array row into a deleted array row.

This is similar to the 4GL procedure ArrayDeleteRow( ).

Chapter 23: Sample 4GL Application

This section contains the following topics:

This chapter presents the complete 4GL code for an application to maintain employee records. Although from the end user's perspective, the application is functionally simple, it incorporates a wide variety of 4GL coding techniques and features.

The 4GL code for the application contains extensive comments to explain the specific operations and statements. Where necessary, the chapter provides additional description. For more information about a 4GL statement or coding technique, see the appropriate section of this guide.

The application contains four frames that are easily constructed with ABF or Vision, and standard forms that have been edited with VIFRED.

Employee Database

The Employee application allows a user to browse and update an Employee database consisting of the following tables:

- Employees
- Employee_histories
- Positions

Each of these tables is described below.

Employees Table

The Employees table contains one row for each employee. The key column is "ssn" (Social Security number). This table contains the following columns:

Column Name	Data Type	Nulls
ssn	i4	no
last_name	varchar(18)	no
first_name	varchar(12)	no

Column Name	Data Type	Nulls
initial	char(1)	no
last_updated_at	date	no
last_updated_by	varchar(32)	no

Employee_ histories Table

The Employee_histories table contains one row for each position held by a specific employee. The key columns are "ssn" and "start_date," the date on which an employee starts at a specific position. This table contains the following columns:

Column Name	Data Type	Nulls
ssn	i4	no
start_date	date	no
position_code	i4	no
salary	money	no

Positions Table

The Positions table contains one row per position code. The key column is "position_code." The Positions table contains the following columns:

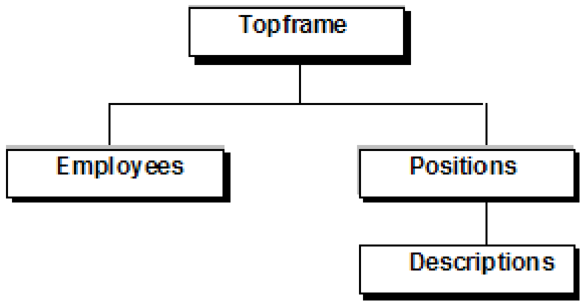
Column Name	Data Type	Nulls
position_code	i4	no
position_title	varchar(32)	no
position_description	varchar(1900)	no
last_updated_at	date	no
last_updated_by	varchar(32)	no

Referential Integrity

The Employee_histories table is joined to the Employees table on "ssn" and to the Positions table on "position_code." The application ensures *referential integrity;* the user is not allowed to specify a Social Security number or position code in a record in the Employee_histories table if there is no corresponding record in the Employees table or Positions table.

Employee Application

The following figure shows the Employee application:



The frames have the following functions:

- Topframe is the top menu frame.
- The Employees frame lets users update the Employees table and the Employee_histories table in a master/detail format.
- The Positions frame lets users update the Positions table.
- The Descriptions frame lets users view the information held in the description field and import the information from or export it to an ASCII file on disk.

The sections below provide the following information about each of the frames:

- The form for the frame
- A brief discussion of the 4GL statements and coding techniques that the frame illustrates
- The full 4GL source code

The application also uses a 4GL procedure called Confirm. A separate section below describes this procedure.

4GL Statements and Coding Techniques

The following table summarizes some of the 4GL statements and coding techniques used by the Employees application. In most cases, the statements and techniques are indicated by comments in the actual code. Some of the features are also described in the sections about specific frames or procedures.

Statement or Technique	Description	Where Used
insert, update, and delete statements	These statements manipulate data in the Employees, Employee_ histories, or Positions table.	Employees and Positions frames
dbmsinfo ('username') function	This function retrieves the name of the current Ingres user as the value to be entered into the "last_updated_by" column of the Employees or Positions table.	Employees and Positions frames
look_up built-in frame with a query	A select statement is used to populate the lookup table with data from the Positions table. The lookup table ensures the validity of a user-specified position code, and obtains the corresponding position description.	Employees frame
look_up built-in frame with an array	The procedure uses an array of a record type (see Global Objects) to populate the confirmation pop-up.	Confirm procedure
length, left, right, and locate functions	These built-in 4GL functions are used to place text within the array.	Confirm procedure
Local procedures	These procedures perform a variety of processing and checking functions for a frame. The code for each procedure is included at the end of the source code file for the relevant frame.	Employees and Positions frames
Local variables	The procedures are described in the declare section of the initialize block of the code, rather	Employees and Positions frames

Statement or Technique	Description	Where Used
	than in the parameter list of the initialize block, to ensure that they are not passed inadvertently as parameters in a callframe or callproc statement.	
Global variables	These variables are used at various places in the application; see Global Objects.	Employees and Positions frames
Table field menu frame	The table field presents user options in the top frame.	Topframe frame
loadtable table field statement	This statement enters into the table field the names and descriptions of the frames the user can call.	Topframe frame
inittable table field statement	This statement makes the table field read only.	Topframe frame
insertrow and deleterow statements	These statements add to or delete from the database any table field rows that the user has updated or deleted, respectively.	Employees and Positions frames
unloadtable table field statement	This statement updates a table with data that a user has saved in a table field.	Employees and Positions frames
Manipulating the display mode of forms and fields	This feature lets you control the user's ability to manipulate data at various points in the application.	Employees and Positions frames
Manipulating the change bit of a form or row	This feature lets the application execute specific code based on whether the user has saved any changed data.	Employees and Positions frames
Submenus	Submenus let the user perform various operations after selecting data.	Employees and Positions frames
resume statement	This statement returns the cursor to the correct field and transfers control after an error has occurred.	Employees and Positions frames
select to simple fields	This statement retrieves data from the Employees table into the	Employees frame

Statement or Technique	**Description**	**Where Used**
	simple fields of the form.	
Master/Detail select	This statement retrieves an Employee record and all corresponding Employee_histories records into the simple fields and table field columns of the form.	Employees frame
select to a complex object	This statement retrieves records into the table field in which the Positions table is displayed.	Positions frame
file access functions	These statements allow data transfer between a field and an ASCII text file on disk.	Descriptions frame
callframe byref	This statement calls a frame and passes the description information by reference.	Positions frame
Qualifications in simple fields	You can let users enter a qualification into any of the fields of the Employees table (except the display-only fields) to display specific records from the Employees table.	Employees frame
Qualifications in a table field	You can let users enter a specific position code or description, or use a pattern-matching or wildcard character in the "Position Description" column, to display specific records from the Positions table.	Positions frame

Global Objects

The Employees application uses the following global objects:

- The global variable "user_name"

  This variable is used in the Employees and Positions frames to hold the value returned by the dbmsinfo ('username') function. This value is the name of the current Ingres user, and is inserted into the "last_updated_by" column of the Employees table or the Positions table (depending on the frame).

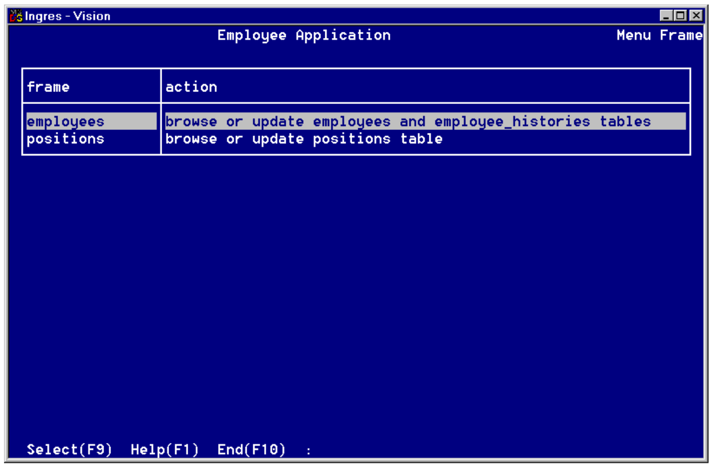
- The record type "choice_line"

  This record type has two attributes: "choice" and "explanation." The record type is used by the Confirm procedure to load values into an array.

  Even though the record type is used by a single procedure in the application, ABF requires that record types be declared globally.

Topframe Frame

The Topframe frame provides a table field menu to allow users to select the Employees frame or the Positions frame. The following figure shows the Topframe frame:



```
Ingres - Vision
                    Employee Application                Menu Frame

 frame      | action
 employees  | browse or update employees and employee_histories tables
 positions  | browse or update positions table

 Select(F9)  Help(F1)  End(F10)  :
```

The table field is named "tf" in the 4GL source code; the table field columns are "frame" and "action."

Menu Operations

Topframe provides the following menu operations:

Menu Item	Function
Select	Calls the frame on whose name the cursor rests
Help	Displays information about this frame
End	Exits the application

4GL Source Code

The following is the UNIX 4GL source code for Topframe. Note the equivalent 4GL source code for other systems is nearly identical to this UNIX example. The only difference is the format of the helpfile pathname.

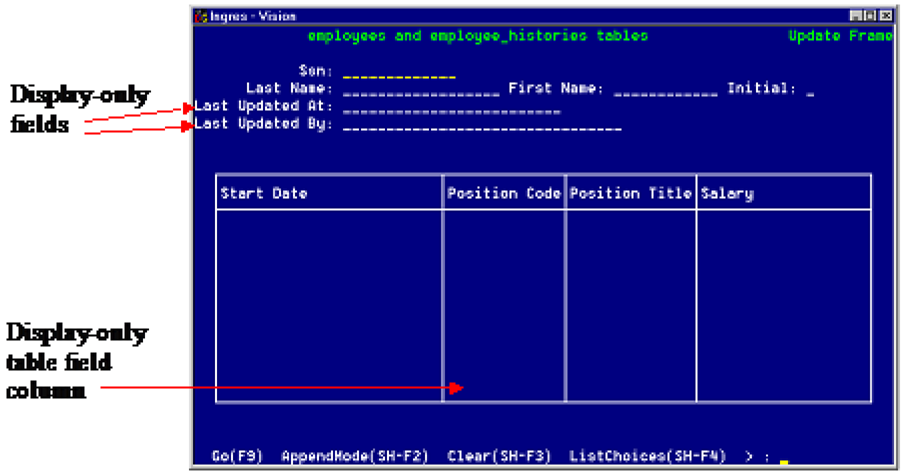
```
/*
** Frame: topframe
** Form: samp_topframe
** Source File: topframe.osq
** Description:
**
** Displays a read-only table field listing actions the user
** may perform; the user chooses one by positioning the cursor
** on the desired action and choosing 'Select'.
**
*/

initialize =
begin
    inittable tf read;
    loadtable tf ( frame = 'employees',
        action = 'browse or update employees and'
        + ' employee_histories tables' );
    loadtable tf ( frame = 'positions',
        action = 'browse or update positions table' );

    scroll tf to 1;
end
```

```
'Select' (explanation = 'Select a command'), key frskey4 =
begin
    callframe :tf[].frame;
end

'Help' (explanation = 'Display help for this frame'), key frskey1 =
begin
    helpfile ''
        '/m/corgi/supp60/employee/employee.hln';
end

'End' (explanation = 'Return to previous frame'), key frskey3 =
begin
    return;
end
```

Employees Frame

The Employees frame uses the Employees table and Employee_histories table in a Master/Detail relationship. A user can display a record (or, by entering a qualification, a set of records) from the Employees table and corresponding records from the Employee_histories table. If desired, the user then can update a selected record or add new records.

The following figure shows the form for the Employees frame:



```
          employees and employee_histories tables              Update Frame

           Ssn: _____________
     Last Name: __________________ First Name: ____________ Initial: _
Last Updated At: _________________________
Last Updated By: ________________________________

  Start Date                | Position Code | Position Title | Salary

  Go(F9)  AppendMode(SH-F2)  Clear(SH-F3)  ListChoices(SH-F4)  > :
```

The form components have the following names:

- The table field is named "tf"
- The table field columns are named "start_date," "position_code," "position_description," and "salary"
- The simple fields are named "ssn," "last_name," "first_name," "initial," "last_updated_at," and "last_updated_by"

The "last_updated_at" and "last_updated_by" columns are display-only. When a row in the Employee_histories table is updated, the "last_updated_at" and "last_updated_by" columns in the corresponding row of the Employees table are updated to indicate when and by whom each row was last updated.

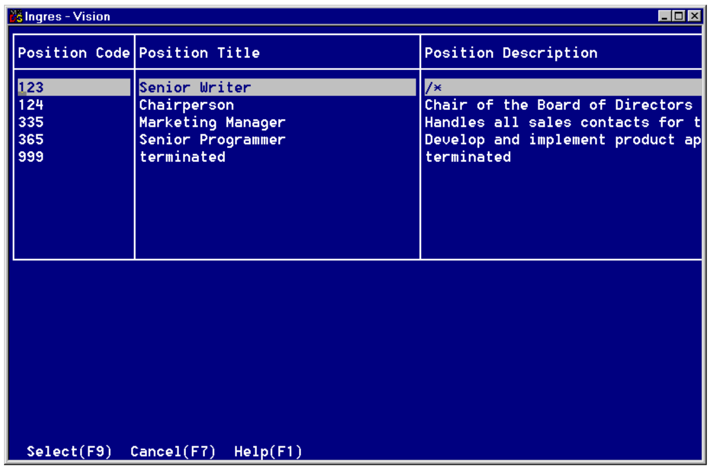
The "last_updated_at" column is used to implement an optimistic locking scheme as follows:

- The DBMS does not hold any locks before the user saves data to the database.
- A commit or rollback statement is issued if needed to release locks before issuing a prompt or message or before exiting from an activation.
- The update and delete statements are coded to fail if another user has changed the same records (as determined by checking the "last_updated_at" column).

The frame uses the Positions table as a lookup table to ensure the validity of a user-specified position_code, and to obtain the corresponding position_title and position_description. A select statement is used to populate the lookup table with data.

When a user selects ListChoices with the cursor in the "position_code" field after having selected data, the window displays a table field containing the valid position codes. The following figure shows this window.

Because the "position_description" column is of type varchar(1900), the table is too wide to display as a pop-up. Therefore, the lookup table is displayed full screen. The column "position_description" is scrollable.



```
Position Code  Position Title       Position Description
123            Senior Writer        /*
124            Chairperson          Chair of the Board of Directors
335            Marketing Manager    Handles all sales contacts for t
365            Senior Programmer    Develop and implement product ap
999            terminated           terminated

 Select(F9)  Cancel(F7)  Help(F1)
```

Menu Operations

The Employees frame provides the following menu operations:

Menu Item	**Function**
Go	Displays a Master record and all corresponding Detail records If the user has entered a qualification into a field of the Master table and then presses Go, displays the first record (in alphabetical order by last name) that meets the qualification. The user then can use the Next operation to page through any additional selected records. After a Master record (and any Detail records) is displayed, the frame is put into update mode and displays a submenu as described in the table Go and AppendMode Submenu Operations.
AppendMode	Puts the frame into append mode and lets the user add new records, and displays a submenu as described in the table Go and AppendMode Submenu Operations.
Clear	Clears all data from the screen
ListChoices	Displays format information about the field on which the cursor is positioned
Help	Displays information about this frame
End	Returns to the previous frame

The following operations are available after a user has selected Go or AppendMode:

Menu Item	Function
RowDelete	Deletes the current row from the table field
RowInsert	Inserts a blank row into the table field so that the user can add a new Detail record
AddNew	Saves a changed Master record and its Detail records (if any) to the database without overwriting the original records
Save (Go submenu only)	Writes the user's changes to the database.
Delete (Go submenu only)	Deletes a Master record and any corresponding Detail records.
Next (Go submenu only)	Displays the next record from the Master table. If the user has entered a qualification, this is the next record that meets the qualification.
ListChoices	When the cursor is positioned on the "Position Code" table field column, displays a list of valid position codes and descriptions from which the user can select. (On other fields or table field columns, ListChoices displays format information about the field or column.)
Help	Displays information about the current frame.
End	Clears the screen and returns the frame to query mode.

4GL Source Code

The following is the complete UNIX 4GL source code file for the Employees frame. The code for the frame's local procedures follows the basic frame processing code.

Note the equivalent 4GL source code for other systems is nearly identical to this UNIX example. The only difference is the format of the helpfile pathnames.

```
/*
** Frame: employees
** Form: samp_employees
** Source File: employees.osq
** Description:
**
** Allows the user to browse and update the Employees and
** Employee_histories tables. The tables are presented to the
** user in Master/Detail form (Employees is the Master, and
** Employee_histories is the detail). The Master records are
** presented in alphabetical order (by last name, first
** name, and initial).
**
** The user is allowed to do arbitrary inserts, updates, and
** deletes on both the master and the details as long as
** referential integrity is maintained. The user can update
** any fields in the tables except for the last_updated_at
** (time) and last_updated_by (user) columns of the Master
** (Employee) table, which are maintained by code in this
** frame.
**
** The last_updated_at column is used to implement an
** optimistic locking scheme: If two users simultaneously
** manipulate a Master record (and its associated details),
** the changes of the first user to select Save are written
** to the database; the second user is informed that his
** changes didn't take.
*/
```

```
initialize =
declare

  /* hidden versions of primary keys & join fields */
  hid_ssn = integer not null,
  tf.hid_start_date = date not null,
  tf.hid_position_code = integer not null,

/* working variables */
  char1 = char(1) not null,          /* holds answer to
                                     ** Yes/No prompts */
  error_no = integer not null,       /* holds DBMS statement
                                     ** error number */
  i = integer not null,              /* general purpose
                                     ** integer */
  obj_name = char(32) not null,      /* holds an object name */
  row_count = integer not null,      /* holds DBMS statement
                                     ** row count */
  row_number = integer not null,     /* holds table field
                                     ** row number (positive when
                                     ** processing the non-deleted
                                     ** rows of a tablefield, zero
                                     ** or negative otherwise) */
  rows_found = char(1) not null,     /* tells if query
                                     ** selected >0 rows */
  row_state = integer not null,      /* holds table field
                                     ** row state */



  save_ok = char(1) not null,        /* tells if Save is legal */
                                     /* the following 2 variables
                                     ** are used by local procedures
                                     ** that print error messages */
  current_op = varchar(16) not null, /* current activation
                                     ** operation */
  record_type = varchar(8) not null, /* set to 'master' or 'detail'
                                     ** during an INSERT, UPDATE,
                                     ** or DELETE; set to ''
                                     ** during other DB stmts */

  /* local procedures */
  do_addnew = procedure returning integer not null,
  do_save = procedure returning integer not null,
  do_delete = procedure returning integer not null,
  check_io_err = procedure returning integer not null,
  do_listchoices = procedure returning none,
  do_after_code = procedure returning integer not null,
  verify_in_tf = procedure returning char(1) not null,
  verify_nothing_to_save = procedure returning char(1) not null,
begin
  set_forms frs (validate(nextfield) = 1,
    validate(previousfield) = 1,
    activate(nextfield) = 1, activate(previousfield) = 1,
    activate(menuitem) = 1, activate(keys) = 1,
    getmessages = 0);

  /* query mode required for qualification function */
  set_forms form (mode = 'query');
  set_forms field samp_employees (mode(tf) = 'read');

  set autocommit off;


  /* If the application hasn't done so yet, the following
  ** statements ask the DBMS for the name of the user
  ** running the application, and save the name in a global
  ** variable.
  */
  if (user_name = '') then
    select user_name = dbmsinfo('username');
    commit work;
  endif;

  row_number = 0;
  record_type = '';
end


'Go' (explanation = 'run query'), key frskey4 =
begin
  current_op = 'Go';

  rows_found = 'n';

  message 'Selecting data . . .';


 /*# begin Select\Master */

  samp_employees := select
    ssn = m.ssn, hid_ssn = m.ssn,
    last_name = m.last_name, first_name = m.first_name,
    initial = m.initial, last_updated_at = m.last_updated_at,
    last_updated_by = m.last_updated_by
  from employees m
  where
  qualification(m.ssn = ssn,
    m.last_name = last_name, m.first_name = first_name,
    m.initial = initial)
  order by last_name asc, first_name asc, initial asc


/*# end Select\Master */
/*# begin Select\Detail */



  samp_employees.tf := repeated select
    start_date = d.start_date, hid_start_date = d.start_date,
    position_code = d.position_code, salary = d.salary,
    position_title = dl.position_title
  from employee_histories d, positions dl
  where
    d.position_code = dl.position_code and d.ssn = :ssn
  order by start_date asc


/*# end Select\Detail */




  begin                 /* begin submenu in 'Go' menuitem */
  initialize =
  begin
    commit work;                 /* release locks */

    /* submenu temporarily changes form from query to
    ** update mode */

    set_forms field samp_employees (mode(tf) = 'fill');

    save_ok = 'y';        /* 'Save' is okay now, because
                          ** the AddNew menuitem has not
                          ** been run on this data.
                          */
    rows_found = 'y';     /* indicate that >0 rows
                          ** qualified */

    set_forms form (change = 0);      /* typing query
                                      ** qualification
                                      ** set change = 1 */
  end


  'RowDelete' (validate = 0, activate = 0,
    explanation = 'Delete current row from table field') =
  begin
    current_op = 'RowDelete';

    if (verify_in_tf() = 'y') then
      deleterow tf;
      set_forms form (change = 1);
    endif;
  end
```

```
'RowInsert' (explanation = 'Open new row in table field') =
begin
  current_op = 'RowInsert';

  if (verify_in_tf() = 'y') then
    validrow tf; /* error if current row invalid */
    inquire_forms table '' (i = rowno);
    insertrow tf[i-1] (_state = 0);
  endif;
end


'AddNew' (activate = 1,
  explanation =
  'Insert current screen data into database') =
begin
  current_op = 'AddNew';

  validate; /* validate all fields on form */

  i = callproc do_addnew; /* attempt to save the data */
  if (i = 0) then
    /* data saved successfully */
    set_forms form (change = 0);
    save_ok = 'n';   /* 'Save' is forbidden now,
                     ** because the AddNew menuitem
                     ** has been run on this data.*/
    mode 'fill';     /* display default values and
                     ** clear simple fields*/
    set_forms form (mode = 'update'); /* cursor off
                                      **Query-only flds*/
    clear field tf;
  else               /* error occurred */
    record_type = '';
    if (row_number > 0) then
      scroll tf to :row_number;
      row_number = 0;
      resume field tf;
    endif;
  endif;
end
```

```
'Save' (activate = 1,
  explanation = 'Update database with current screen data'),
  key frskey8 (activate = 1) =
begin
  current_op = 'save';

  /* Must prevent AddNew followed by Save or Delete.
  ** Reasons: 1. User may have changed keys before
  ** selecting AddNew. Save/Delete assume hidden
  ** field/column versions of keys give true database
  ** identity of the displayed data.
  ** 2. Table field row _STATEs won't show just the
  ** changes made since AddNew.
  */
  if (save_ok = 'n') then
    callproc beep; /* 4gl built-in procedure */
    message 'Error: You cannot Save changes to data if'
      + ' you have previously selected AddNew on'
      + ' that data. Changes not Saved. To change'
      + ' this data you must reselect it, make'
      + ' changes and then press Save.'
      with style = popup;
    resume;
  endif;


  inquire_forms form (i = change);
  if (i = 1) then
    validate; /* validate all fields on form */

    i = callproc do_save; /* attempt to save the data */
    if (i = 0) then
      /* data saved successfully */
      set_forms form (change = 0);

      next;           /* Changes saved. Get next
                      ** screen of data. */
      commit work;    /* Release any locks acquired
                      ** while selecting data for
                      ** the "next" statement.
                      */
    else /* error occurred */
      record_type = '';
      if (row_number > 0) then
        scroll tf to :row_number;
        row_number = 0;
        resume field tf;
      endif;
    endif;
  else
    message 'No changes to Save.' with style = popup;
  endif;
end
'Delete' (validate = 0, activate = 0,
  explanation =
  'Delete current screen of data from Database') =
begin
  current_op = 'Delete';


  /* Must prevent AddNew followed by Save or Delete.
  ** Reasons:1. User may have changed keys before
  ** selecting AddNew.Save/Delete assume hidden
  ** field/column versions of keys give
  ** true database identity of the displayed data.
  ** 2. Table field row _STATEs won't show just the
  ** changes made since AddNew.
  */
  if (save_ok = 'n') then
    callproc beep; /* 4gl built-in procedure */
    message 'Error: You cannot Delete data if you have'
      + ' previously selected AddNew on that data.'
      + ' Data not Deleted. To Delete this data you'
      + ' must reselect it and then press Delete.'
      with style = popup;
    resume;
  endif;


  char1 = callproc confirm (
    question = 'Delete this master and all its details'
              + ' from the Database?',
    no = 'Cancel the "Delete" operation.',
    yes = 'Delete this master and all its details.'
  );
  if (char1 = 'n') then
    resume;
  endif;
  i = callproc do_delete; /* attempt to delete the data */
  if (i = 0) then
    /* data deleted successfully */
    set_forms form (change = 0);

    next;          /* Data deleted. Get next
                   ** screen of data */
    commit work;   /* Release any locks acquired
                   ** while selecting data for
                   ** the "next" statement.
                   */
  else             /* error occurred */
    record_type = '';
  endif;
end


'Next' (validate = 0, activate = 0,
  explanation = 'Display next row of selected data'),
  key frskey4 (validate = 0, activate = 0) =
begin
  current_op = 'Next';

  if (verify_nothing_to_save() = 'n') then
    resume menu;
  endif;
  set_forms form (change = 0);
  next;
  commit work;    /* Release any locks acquired
                  ** while selecting data for
                  ** the "Next" statement.
                  */
  save_ok = 'y';  /* 'Save' is okay now, because
                  ** the AddNew menuitem has not
                  ** been run on this data.
                  */
end
```

```
'ListChoices' (validate = 0, activate = 0,
  explanation = 'Show valid values for current field'),
  key frskey10 (validate = 0, activate = 0) =
begin
  current_op = 'ListChoices';

  callproc do_listchoices;
end



key frskey5 (explanation = 'Scroll to top of table field') =
begin
  current_op = 'Top';

  if (verify_in_tf() = 'y') then
    scroll tf to 1;
  endif;
end


key frskey6 (explanation = 'Scroll to bottom of table field') =
begin
  current_op = 'Bottom';

  if (verify_in_tf() = 'y') then
    scroll tf to end;
  endif;
end


key frskey7
(explanation = 'Search table field for a specified value') =
begin
  current_op = 'FindRecord';

  i = callproc find_record;
end
```

```
'Help' (validate = 0, activate = 0,
  explanation = 'Display help for this frame'),
  key frskey1 (validate = 0, activate = 0) =
begin
  helpfile 'employees and employee_histories tables'
    '/m/corgi/supp60/employee/employees.hlp';
end

'End' (validate = 0, activate = 0,
  explanation = 'Return from Update mode to Query mode'),
  key frskey3 (validate = 0, activate = 0) =
begin
  current_op = 'End';

  if (verify_nothing_to_save() = 'n') then
    resume menu;
  endif;
  endloop; /* exit submenu */
end

after field 'tf.position_code' =
begin
  if (do_after_code() = 1) then
    resume;
  endif;
  resume next;
end
end; /* end of submenu in 'Go' menuitem */
```

```
  /* display mode reverts to prior mode ('query') after submenu */

  if (rows_found = 'y') then
    set_forms field samp_employees (mode(tf) = 'read');
    clear field all;
    set_forms form (change = 0);
  endif;
end /* end of 'Go' menuitem */


'AppendMode' (validate = 0, activate = 0,
  explanation = 'Display submenu for Appending new data') =
begin
  set_forms form (mode = 'update');
  set_forms field samp_employees (mode(tf) = 'fill');

  save_ok = 'n';   /* 'Save' is not okay now;
                   ** it's not on menu */
  display submenu
  begin

  'RowDelete' (validate = 0, activate = 0,
    explanation = 'Delete current row from table field') =
  begin
    current_op = 'RowDelete';

    if (verify_in_tf() = 'y') then
      deleterow tf;
      set_forms form (change = 1);
    endif;
  end


  'RowInsert' (explanation = 'Open new row in table field') =
  begin
    current_op = 'RowInsert';

    if (verify_in_tf() = 'y') then
      validrow tf; /* error if current row invalid */
      inquire_forms table '' (i = rowno);
      insertrow tf[i-1] (_state = 0);
    endif;
  end
```

```
'AddNew' (activate = 1,
  explanation = 'Insert current screen data into database') =
begin
  current_op = 'AddNew';

  validate; /* validate all fields on form */

  i = callproc do_addnew; /* attempt to save the data */
  if (i = 0) then
    /* data saved successfully */
    set_forms form (change = 0);

    mode 'fill';   /* display default values
                   ** and clear simple fields*/
    set_forms form (mode = 'update'); /* cursor off
                                      ** query-only fields*/
    clear field tf;
  else             /* error occurred */
    record_type = '';
    if (row_number > 0) then
      scroll tf to :row_number;
      row_number = 0;
      resume field tf;
    endif;
  endif;
end


'ListChoices' (validate = 0, activate = 0,
  explanation = 'Show valid values for current field'),
  key frskey10 (validate = 0, activate = 0) =
begin
  current_op = 'ListChoices';

  callproc do_listchoices;
end



key frskey5 (explanation = 'Scroll to top of table field') =
begin
  current_op = 'Top';

  if (verify_in_tf() = 'y') then
    scroll tf to 1;
  endif;
end
```

```
key frskey6 (explanation = 'Scroll to bottom of table field') =
begin
  current_op = 'Bottom';

  if (verify_in_tf() = 'y') then
    scroll tf to end;
  endif;
end



key frskey7
(explanation = 'Search table field for a specified value') =
begin
  current_op = 'FindRecord';

  i = callproc find_record;
end


'Help' (validate = 0, activate = 0,
  explanation = 'Display help for this frame'),
  key frskey1 (validate = 0, activate = 0) =
begin
  helpfile 'employees and employee_histories tables'
    '/m/corgi/supp60/employee/employees.hla';
end


'End' (validate = 0, activate = 0,
  explanation = 'Leave AppendMode and re-enter Query mode'),
  key frskey3 (validate = 0, activate = 0) =
begin
  current_op = 'End';

  if (verify_nothing_to_save() = 'n') then
    resume menu;
  endif;
  endloop;            /* exit submenu */
end
```

```
  after field 'tf.position_code' =
  begin
    if (do_after_code() = 1) then
      resume;
    endif;
    resume next;
  end
  end;  /* end of submenu in 'AddNew' menu item */

  set_forms form (mode = 'query');
  set_forms field samp_employees (mode(tf) = 'read');
  clear field all;
end

'Clear' (validate = 0, activate = 0,
  explanation = 'Clear all fields') =
begin
  clear field all;
end



'ListChoices' (validate = 0, activate = 0,
  explanation = 'Show valid values for current field'),
  key frskey10 (validate = 0, activate = 0) =
begin
  current_op = 'ListChoices';

  callproc do_listchoices;
end

'Help' (validate = 0, activate = 0,
  explanation = 'Display help for this frame'),
  key frskey1 (validate = 0, activate = 0) =
begin
  helpfile 'employees and employee_histories tables'
    '/m/corgi/supp60/employee/employees.hlq';
end
```

```
'end' (validate = 0, activate = 0,
  explanation = 'Return to previous frame'),
  key frskey3 (validate = 0, activate = 0) =
begin
  return;
end

/*
** Local procedure: do_addnew
**
** Description:
**       Attempts to add the data in the current screen to the
**       database.
**
**       Note: the caller must validate the data in the screen
**       before calling this procedure.
**
** Returns:
**       0 if data saved successfully.
**       1 if an error occurred.
*/


procedure do_addnew =
begin
  message 'Saving new data . . .';

  record_type = 'master';

  repeated insert
  into employees(ssn, last_name, first_name, initial,
    last_updated_at, last_updated_by)
  values(ssn, last_name, first_name, initial,
    'now', user_name);

  if (check_io_err() != 0) then
    return 1;
  endif;
```

```
record_type = 'detail';
unloadtable tf (row_state = _state, row_number = _record)
begin
  /* insert new, unchanged & changed rows */
  if ((row_state = 1) or (row_state = 2) or (row_state = 3)) then

    /* Try to insert the new detail record into
    ** employee_histories. Instead of using a
    ** straightforward VALUES clause,
    ** use an artificial subselect that is equivalent
    ** to a VALUES clause, except that it will insert
    ** nothing if the position_code in the new detail
    ** record has been deleted from the positions
    ** table since it was loaded into the tablefield
    ** (or entered by the user).
    */
    repeated insert
    into employee_histories(ssn, start_date,
      position_code, salary)
    select :ssn, :tf.start_date,
      :tf.position_code, :tf.salary
    from positions dl
    where dl.position_code = :tf.position_code;
    if (check_io_err() != 0) then
      return 1;
    endif;


    inquire_sql (row_count = rowcount);
    if row_count <= 0 then
      rollback work;
      message 'The "Save" operation was not '
        + ' performed, because the position_code'
        + ' in a detail record has been deleted'
        + ' from the positions table by another'
        + ' user while you were updating the'
        + ' tablefield. The cursor will be'
        + ' placed on the row where the error'
        + ' occurred. Tab to Position Code and'
        + ' select ListChoices to see which'
        + ' position codes are now available.'
        with style = popup;
      return 1;
    endif;
  endif;
end;
```

```
  row_number = 0;
  record_type = '';

  commit work;

  if (check_io_err() != 0) then
    return 1;
  endif;

  return 0;
end /* end of do_addnew */


/*
** Local procedure: do_save
**
** Description:
**      Attempts to update the database using the data in the
**      current screen.
**
**      Note: the caller must verify that the form contains
**      changed data and validate it before calling this procedure.
**
** Returns:
**       0 if data saved successfully.
**       1 if an error occurred.
*/
procedure do_save =
declare
  need_check = char(1) not null,
begin
  message 'Saving changes . . .';


  /* The logic below may require modification in case of rules
  ** or referential integrities on master data. For example,
  ** if a master update changes the value of join field,
  ** and that fires a rule that changes the join key value
  ** for all matching detail data, then that could cause
  ** detail updates to fail (rowcount=0).
  */
  record_type = 'master';

  repeated update employees
  set ssn = :ssn,
    last_name = :last_name, first_name = :first_name,
    initial = :initial, last_updated_at = 'now',
    last_updated_by = :user_name
  where ssn = :hid_ssn and last_updated_at = :last_updated_at;

  if (check_io_err() != 0) then
    return 1;
  endif;
```

```
inquire_sql (row_count = rowcount);
if row_count <= 0 then
  rollback work;
  message 'The "Save" operation was not performed,'
    + ' because the master record (or one of its details)'
    + ' has been updated or deleted'
    + ' by another user since you selected it.'
    + ' Before selecting "Next" or "End", you may wish'
    + ' to (1) make a note of your changes,'
    + ' or (2) change the key (ssn) and select "AddNew".'
    + ' In either case, you should subsequently'
    + ' (1) determine what happened to the record'
    + ' (e.g. by attempting to re-select it)'
    + ' and (2) reconcile your changes'
    + ' with the other users'' changes.'
    with style = popup;
  save_ok = 'n';
  return 1;
endif;


record_type = 'detail';

/* Process the deleted rows before other rows.
** If we process deleted rows last, which is the order
** that Unloadtable delivers them in, then if
** a row with an identical key is deleted and then inserted
** into the table field before the user selects Save,
** we will erroneously attempt the insert before the
** delete, and the insert will fail (duplicate key).
*/
unloadtable tf (row_state = _state, row_number = _record)
begin
  if (row_state = 4) then /* deleted */

    /* delete row using hidden field keys in where clause. */
    repeated delete from employee_histories
    where ssn = :hid_ssn and start_date = :tf.hid_start_date;

    if (check_io_err() != 0) then
      return 1;
    endif;
  endif;
end; /* end of first unloadtable */
```

```
/* process all but Deleted rows */
unloadtable tf (row_state = _state, row_number = _record)
begin
  need_check = 'n';

  if (row_state = 1) then /* new */

    /* Try to insert the new detail record into
    ** employee_histories. Instead of using a
    ** straightforward VALUES clause,
    ** use an artificial subselect which is equivalent
    ** to a VALUES clause, except that it inserts nothing
    ** if the position_code in the new detail record
    ** has been deleted from the positions table since
    ** it was loaded into the tablefield (or entered
    ** by the user).
    */

    repeated insert
    into employee_histories(ssn, start_date,
      position_code, salary)
    select :ssn, :tf.start_date,
      :tf.position_code, :tf.salary
    from positions dl
    where dl.position_code = :tf.position_code;

    need_check = 'y';


  elseif (row_state = 3) /* table field data changed */
    or (row_state = 2 and
    ssn != hid_ssn) /* join field changed */
  then

    /* Try to update the detail record in
    ** employee_histories. Use the hidden version of
    ** the key field in the WHERE clause.
    ** Also add an artificial condition to the WHERE
    ** clause to ensure that no rows are updated
    ** if the position_code in the detail record
    ** has been deleted from the positions table since
    ** it was loaded into the tablefield (or entered
    ** by the user).
    */
    repeated update employee_histories d
    from positions dl
    set ssn = :ssn, start_date = :tf.start_date,
      position_code = :tf.position_code,
      salary = :tf.salary
    where d.ssn = :hid_ssn
    and d.start_date = :tf.hid_start_date
    and dl.position_code = :tf.position_code;

    need_check = 'y';
  endif;
```

```
    if (need_check = 'y') then

      if (check_io_err() != 0) then
        return 1;
      endif;

      inquire_sql (row_count = rowcount);
      if row_count <= 0 then
        rollback work;
        message 'The "Save" operation was not'
          + ' performed, because the position_code'
          + ' in a detail record has been deleted'
          + ' from the positions table by another'
          + ' user while you were updating'
          + ' the tablefield.'
          + ' The cursor will be placed on the row'
          + ' where the error occurred.'
          + ' Tab to Position Code and'
          + ' select ListChoices to see which'
          + ' position codes are now available.'
          with style = popup;
        return 1;
      endif;
    endif;
  end; /* end of second unloadtable */


  row_number = 0;
  record_type = '';

  commit work;

  if (check_io_err() != 0) then
    return 1;
  endif;

  return 0;
end /* end of do_save */
/*
** Local procedure: do_delete
**
** Description:
**       Attempts to delete the master in the current screen
**       and all its details from the database.
**
**       Note: the caller is responsible for any prompting
**       (e.g. 'Are you sure you want to delete').
**
** Returns:
**       0 if data deleted successfully.
**       1 if an error occurred.
*/
```

```
procedure do_delete =
begin
  message 'Deleting . . .';

  record_type = 'master';

  repeated delete from employees
  where ssn = :hid_ssn and last_updated_at = :last_updated_at;

  if (check_io_err() != 0) then
    return 1;
  endif;


  inquire_sql (row_count = rowcount);
  if row_count <= 0 then
    rollback work;
    message 'The "Delete" operation was not performed,'
      + ' because the master record (or one of its'
      + ' details) has been updated or deleted'
      + ' by another user since you selected it.'
      + ' After selecting "Next" or "End", you may wish'
      + ' to determine what happened to the master'
      + ' record (e.g. by trying to re-select it).'
      with style = popup;
    save_ok = 'n';
    return 1;
  endif;

  record_type = 'detail';

  repeated delete from employee_histories
  where ssn = :hid_ssn;

  if (check_io_err() != 0) then
    return 1;
  endif;

  record_type = '';

  commit work;

  if (check_io_err() != 0) then
    return 1;
  endif;

  return 0;
end /* end of do_delete */
```

```
/*
** Local procedure: check_io_err
**
** Description:
**       Checks to see if the last database I/O statement
**       executed properly; does a ROLLBACK and puts out an
**       error message if not. Note that if an error *has*
**       occurred, the DBMS has already issued its own
**       error message.
**
** Returns:
**       The error number generated by the database I/O statement
**       (0 if no error).
*/



procedure check_io_err =
declare
  err_data_loc = varchar(80) not null,
  cursor_msg = varchar(80) not null,
  correct_msg = varchar(80) not null,
begin
  inquire_sql (error_no = dbmserror);
  if (error_no = 0) then
    return 0;
  endif;
  if (record_type = '') then
    err_data_loc = '';
  else
    err_data_loc = ' The error occurred on a ' +
                  record_type + ' record.';
  endif;
  if (row_number <= 0) then
    cursor_msg = '';
  else
    cursor_msg = ' The cursor will be placed on the row'
      + ' where the error occurred.';
  endif;

  if (error_no = 4700) then   /* deadlock (DBMS has
                              ** already done ROLLBACK) */
    correct_msg = '';         /* nothing to fix on a
                              ** deadlock; just retry */
  else
    correct_msg = ' correct the error (if possible) and';
    rollback work;
  endif;
  Message 'the "' + current_op + '" Operation Was Not Performed,'
    + ' due to the error described in the previous message.'
    + err_data_loc + cursor_msg
    + ' Please' + correct_msg + ' select "' + current_op'
    + '" again.'
    with style = popup;
  return error_no;3
end
```

```
/*
** Local procedure: do_listchoices
**
** Description:
**      Implements the ListChoices activation for the main menu
**      and both submenus.**
** Returns: none.
*/
procedure do_listchoices =
declare
  value_selected = integer not null,   /* if > 0, indicates value
                                       ** selected onto form */
begin
  value_selected = 0;
  inquire_forms field '' (obj_name = name);
  if (obj_name = 'tf') then /* cursor in table field */

    /* Skip look_up call if table field is empty
    ** (Returns an error if you read from or assign
    ** to an empty table field)
    */
    inquire_forms table '' (i = datarows('tf'));
    if (i >0) then
      inquire_forms table '' (obj_name = column);
      if (obj_name = 'position_code') then

        value_selected = callframe look_up (
          ii_rows = 10;
          ii_query = select distinct position_code,
                         position_title, position_description
                     from positions
                     order by position_code,
                       position_title;
          ii_field1 = 'position_code';
          ii_field2 = 'position_title';
          ii_field3 = 'position_description';
          ii_titles = 1;
          ii_field_title1 = 'Position Code';
          ii_field_title2 = 'Position Title';
          ii_field_title3 = 'Position Description';
          position_code = byref(tf.hid_position_code);
          position_title =
            byref(tf.position_title)
        );
        if (value_selected > 0) then
          tf.position_code =
            tf.hid_position_code;
          commit work; /* release shared locks on
                       ** lookup table */
        endif;
      endif;
    endif;
  endif;
  if (value_selected <= 0) then
    /* No look_up available for current field */
    value_selected = callproc help_field;
  endif;

  if (value_selected > 0) then
    /* value was selected onto form */
    set_forms form (change = 1);
  endif;

  return;
end


/*
** Local procedure: do_after_code
**
** Description:
**       Implements the AFTER FIELD 'tf.position_code'
**       activation for the main menu and both submenus.
**
**       If the field has changed and the form is not in query
**       mode, the position_title field is derived from
**       the position_code field via the positions table. If
**       the position_code is not found in the positions table,
**       an error is displayed (indicating that the
**       position_code is invalid).
**
** Returns:
**       0 if tf.position_code is valid
**       1 if tf.position_code is invalid
*/
```

```
procedure do_after_code =

begin
  inquire_forms row '' '' (i = change);
  inquire_forms form (obj_name = mode);
  if (i = 1) and (uppercase(obj_name) != 'QUERY') then

    samp_employees := repeated select
      :tf[].position_code = positions.position_code,
      :tf[].position_title =
        positions.position_title
    from positions
    where positions.position_code = :tf.position_code;

    inquire_sql (row_count = rowcount);
    if row_count <= 0 then
      callproc beep; /* 4gl built-in function */
      message '"' + ifnull(varchar(:tf.position_code), '') +
        '" is not a valid value for this field' +
        ' (select ListChoices for help).'
        with style = popup;
      set_forms row '' '' (change = 1);
      return 1;
    endif;
    commit work;     /* release shared locks on
                     ** lookup table */
  endif;
  return 0;
end


/*
** Local procedure: verify_in_tf
**
** Description:
**       Checks to see if cursor is positioned in the table
**       field; puts out error message if not.
**
** Returns:
**       'y' if in the table field.
**       'n' if not in the table field.
*/
procedure verify_in_tf =
begin
  inquire_forms field '' (i = table);
  if (i = 1) then
    return 'y';
  endif;
  callproc beep; /* 4gl built-in procedure */
  message 'You can only "' + current_op +
    '" when your cursor is in a table field.'
    with style = popup;
  return 'n';
end
```

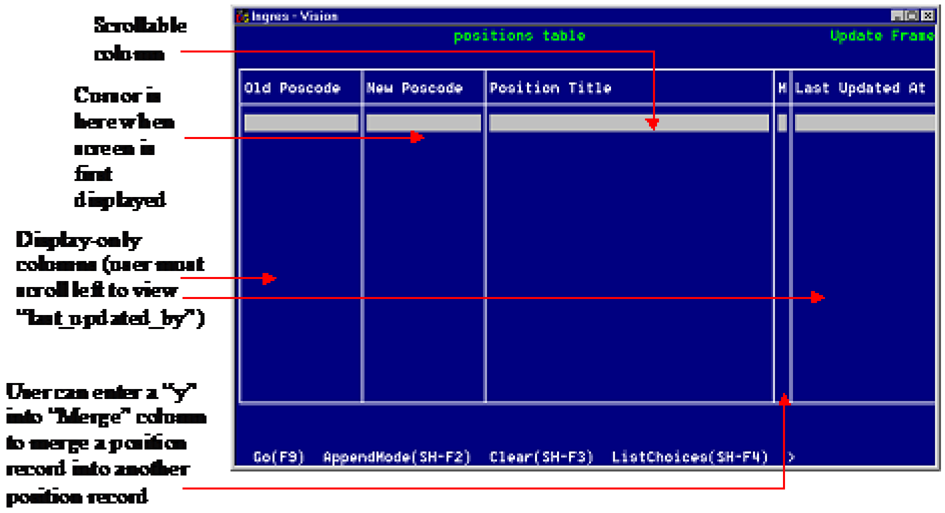
```
/*
** Local procedure: verify_nothing_to_save
**
** Description:
**       Checks to see if anything has changed on the form
**       (and is thus a candidate for a Save). If so, the user
**       is prompted as to whether these changes
**       should be saved. The prompting is done via the
**       procedure "confirm".
**
** Returns:
**       'y' if nothing has changed, or if the user
**       says to discard the changes.
**       'n' if something has changed, and the user
**       says the changes must be saved.
*/
procedure verify_nothing_to_save =
declare
  save_op = varchar(8) not null,
begin
  inquire_forms form (i = change);
  if (i = 0) then
    return 'y';
  endif;

  if (save_ok = 'y') then
    save_op = ' Save or';
  else
    save_op = '';
  endif;

  return confirm (
    question = 'Do you wish to "' + current_op +
      '" without saving changes?',
    no = 'Cancel the "' + current_op + '" operation.' +
      ' (You can then save your changes by selecting' +
      save_op + ' AddNew).',
    yes = '"' + current_op + '" without saving changes.'
  );
end
```

Positions Frame

The Positions frame displays the Positions table (or a user-specified subset) in a table field format. Users can view and update records from the table.

When records are loaded into the table field, the position code is displayed in both the "Old PosCode" and "New PosCode" table field columns. Users cannot modify the old position code; it is displayed to help users keep track of the changes that occur when they select Save (see the section Menu Operations).

The following figure shows the form for the Positions frame:


Scrollable column
Cursor is here when screen is first displayed
Display-only columns (user must scroll left to view "last_updated_by")
User can enter a "y" into "Merge" column to merge a position record into another position record
Ingres - Vision
positions table
Update Frame
Old Poscode
New Poscode
Position Title
M
Last Updated At
Go(F9) AppendMode(SH-F2) Clear(SH-F3) ListChoices(SH-F4) &gt;

The form components have the following names:

- The table field is named "tf"
- The table field columns are named "old_position_code," "position_code," "position_title," "merge," "last_updated_at," "last_updated_by," and a hidden field, "position_description."

Menu Operations

The Positions frame provides the following menu operations:

Menu Item	Function
Go	Displays all records from the Positions table that meet the qualifications that the user has entered into the table field. After the records are displayed, the frame is put into update mode, and displays a submenu as described below.
AppendMode	Puts the frame into append mode and lets the user add new records, and displays a submenu as described below.
Clear	Clears all data from the screen.
ListChoices	Provides information about the format of the field on which the cursor is positioned.
Help	Displays information about this frame.

Menu Item	Function
End	Returns to the previous frame.

The following operations are available after a user has selected Go or AppendMode:

Menu Item	Function
RowDelete	Deletes the current row from the table field (if the row has been added with the RowInsert operation, and has not yet been saved).
RowInsert	Inserts a blank row into the table field so that the user can add a new Detail record.
Reselect	Restores a record from the database, based on the value of the "Old PosCode" column. This operation is useful if a Save operation has failed because another user has updated the record in the database after it was selected into the current user's table field.
MoreInfo	Calls the frame "Descriptions" which allows users to view and edit the position_description field. The "Descriptions" frame also allows users to manipulate the information in the position_description field by importing it from or exporting it to an ASCII file.
Save	Writes the user's changes to the database.
ListChoices	Displays information about the format of the field on which the cursor is positioned.
Help	Displays information about the current frame.
End	Returns to the previous frame.

Deleting Records from the Positions Table

Users cannot use the RowDelete operation to delete a table field row that was loaded from the database. A user can take either of the following actions to delete a record from the Positions table:

- Enter blanks into the "New PosCode" table field column and select Save. The record then is deleted from the Positions table if there are no records in the Employee_ histories table that contain the same position code. If there are any such records, the Save fails.

  This validation is necessary to ensure referential integrity. If the Save does fail, users must use the method described immediately below.

- "Merge" the record as follows:

  a. Change the value in the "New PosCode" column to another valid position code in the Positions table.

  b. Enter y into the "M" (merge) column.

  When the user selects Save, the record is deleted from the Positions table. All records in the Employee_ histories table that have this position code are updated to contain the value entered into the "New PosCode" column.

  To ensure referential integrity, the Save operation fails if there is no record in the Positions table with the value that the user has entered into the "New PosCode" column.

Updating Records in the Positions Table

Users can use the Positions frame to update records in the Positions table as follows:

- Changing a position description by entering a new value into the "position_description" table field column and selecting Save
- Changing the position code by entering a new value into the "position_code" table field column and selecting Save, leaving the "M" blank (or entering n)

  In this case, the application takes the following actions:

  – Ensuring referential integrity by updating all records in the Employee_histories table that contain the position code that was changed

  – Rejecting the change if the value entered for the new position code already exists in the Positions table (because position code is a unique key for the table)

Inserting Records into the Positions Table

Users can use the Positions frame to insert records into the Positions table in either of the following ways:

- By selecting InsertRow and typing data into the resulting blank row
- By scrolling down to the blank row below the last row displayed and typing in new data

Users can insert records in any order.

Calling the Descriptions Frame

Users can select MoreInfo to call the Descriptions frame to view and edit the position description field.

The MoreInfo operation passes the position_description information for the current row by reference to the Descriptions frame. The MoreInfo operation also passes the position_code, position_title, last_updated_at, and last_updated_by fields for the current row to simple read-only fields on the Descriptions form.

The Descriptions frame allows users to manipulate the information in the position_description field by importing it from or exporting it to an ASCII file. Any changes made to the position_description field are passed back to the Positions frame. However, the position_description field is not changed in the database until the record is saved.

See Descriptions Frame (see page 1247) for more information.

4GL Source Code

The following is the complete 4GL source code file for the Positions frame. The code for the frame's local procedures follows the basic frame processing code.

```
/*
** Frame: positions
** Form: samp_positions
** Source File: positions.osq
** Description:
**
** Allows the user to browse and update the Positions table.
** The table is presented to the user in tablefield form.
**
** The user is allowed to do arbitrary inserts, updates, and
** deletes as long as referential integrity is maintained
** (position_code appears as a foreign key in the
** employee_histories table). The user can insert a record by
** selecting InsertRow and typing data into the resulting
** blank row, or by scrolling down to the blank row below
** the last row displayed and typing in new data.
** Records can be inserted in any order.
** The user can delete a record by blanking out its position
** code (in the column labelled New PosCode).
 ** The user can update any fields in the positions table
** except for the last_updated_at (time) and last_updated_by
** (user) columns, which are maintained by code in this frame.
**


** The last_updated_at column is used to implement an
** optimistic locking scheme: If two users simultaneously
** manipulate a record in the Positions table, the changes
** done by the first user to select Save are written to the
** database; the second user is informed that his changes
** didn't take.
**
** If the user changes the position_code in a record in the
** Positions table, all records in the employee_histories
** table with the same position_code are automatically
** updated. The last_updated_at and last_updated_by
** columns of the corresponding master record (in the
** employees table) are also updated.
**
```

```
** A special mechanism is available to "merge" two records in
** the positions table. Merging record A into record B
** entails the following:
** (1) the record in the positions table with position_code A
** is deleted, and
** (2) any records in the employee_histories table with
** position_code A have their position_code changed to B
** (which must already exist in the positions table).
** The user specifies that record A is to be merged into B
** by selecting A into the tablefield, changing the
** "New PosCode" column from A to B, and setting the "M"
** column to 'y'.
**
** There is one hidden field "position_description" in the
** tablefield. Information contained in this field will be
** saved along with the rest of the tablefield to the database.
** The user must select "MoreInfo" (which puts the user into
** the description frame) to view or access the
** position_description information. Any changes made by the user
** in the description frame will be saved on the position frame.
*/



initialize =
declare

  /* working variables*/
  char1 = char(1) not null,       /* holds answer to
                                  ** yes/no prompts */
  error_no = integer not null,    /* holds DBMS
                                  ** statement error no. */
  i = integer not null,           /* general purpose integer */
  i_descript  = integer not null, /* holds change state of
                                  ** description */
  row_count = integer not null,   /* holds DBMS
                                  ** statement row count */
  row_number = integer not null,  /* holds table field
                                  ** row number */
  row_state = integer not null,   /* holds table field
                                  ** row state */

  /* local procedures */
  display_submenu = procedure returning none,
  check_io_err = procedure returning integer not null,
begin
  set_forms frs (validate(nextfield) = 1,
    validate(previousfield) = 1,
    activate(nextfield) = 1, activate(previousfield) = 1,
    activate(menuitem) = 1, activate(keys) = 1,
    getmessages = 0);

  /* query mode required for qualification function */
  set_forms form (mode = 'query');
  set_forms field samp_positions (mode(tf) = 'query');
  set_forms column samp_positions tf (displayonly(merge) = 1);

  set autocommit off;

  /* If the application has not already done so, ask the
  ** DBMS for the name of the user running the application,
  ** and save the name in a global variable.
  */
  if (user_name = '') then
    select user_name = dbmsinfo('username');
    commit work;
  endif;
end


'Go' (explanation = 'Run query'), key frskey4 =
begin
  message 'Selecting data . . .';

  /* selecting to table field with qualification function
  ** changes table field mode to Update, even if 0 rows
  ** were selected.
  */


/*# begin Select\Master */


  samp_positions.tf := select
    old_position_code = m.position_code,
    position_code = m.position_code,
    position_title= m.position_title,
    position_description = m.position_description,
    merge = ' ',  last_updated_at = m.last_updated_at,
    last_updated_by = m.last_updated_by
  from positions m
  where
  qualification(m.position_code = tf.position_code,
    m.position_title = tf.position_title)
  order by position_code asc;
```

```
/*# END Select\Master */

  inquire_sql (row_count = rowcount);
  if (row_count <= 0) then
    rollback work;
    set_forms field samp_positions (mode(tf) = 'query');
    set_forms column samp_positions tf
      (displayonly(merge) = 1);
    message 'No rows retrieved';
    sleep 2;
    resume;
  endif;

  /* Assertion: rows selected above */
  commit work; /* release locks */

  callproc display_submenu(menu_mode = 'u');
end



'AppendMode' (validate = 0, activate = 0,
  explanation = 'Display submenu for Appending new data') =
begin
  callproc display_submenu(menu_mode = 'a');
end




'Clear' (validate = 0, activate = 0,
  explanation = 'Clear all fields') =
begin
  clear field all;
end



'ListChoices' (validate = 0, activate = 0,
  explanation = 'Show valid values for current field'),
  key frskey10 (validate = 0, activate = 0) =
begin
  i = callproc help_field;
end



'Help' (validate = 0, activate = 0,
  explanation = 'Display help for this frame'),
  key frskey1 (validate = 0, activate = 0) =
begin
  helpfile 'positions table'
    '/m/corgi/supp60/employee/positions.hlq';
end
```

```
'End' (validate = 0, activate = 0,
  explanation = 'Return to previous frame'),
  key frskey3 (validate = 0, activate = 0) =
begin
  return;
end


/*
** Local procedure: display_submenu
**
** Description:
**       Displays a submenu suitable for either append mode or
**       update mode.
**
** Input parameter:
**       menu_mode 'a' (append) or 'u' (update).
**
** Returns: none.
*/
procedure display_submenu (
  menu_mode = char(1), /* a/u: append mode or
                        ** update mode? */
) =




declare
  delete_position = char(1),      /* y/n: delete record
                                  ** from Positions? */
  update_position = char(1),      /* y/n: update record
                                  ** in Positions? */
  verify_position = char(1),      /* y/n: check to make
                                  ** sure the record in
                                  ** Positions being
                                  ** merged into really
                                  ** exists */
  update_employees = char(1),     /* y/n: update records
                                  ** in Employees and
                                  ** Employee_histories
                                  ** to reflect change
                                  ** in position_code?
                                  */
  verify_no_employees = char(1),  /* y/n: check to make
                                  ** sure there are
                                  ** no records in
                                  ** Employee_histories
                                  ** with the position
                                  ** code being deleted*/
```

```
begin
  set_forms form (change = 0);    /* typing query
                                  ** qualification
                                  ** set change = 1 */
  set_forms form (mode = 'update');
  /* If table field in query mode (as it will be for an
  ** 'Append'), then the following statement will clear it */
  set_forms field samp_positions (mode(tf) = 'fill');
  set_forms column samp_positions tf (displayonly(merge) = 0);

  display submenu
  begin


  'RowDelete' (validate = 0, activate = 0,
    explanation = 'Delete current row from table field') =
  begin
    if (tf.old_position_code is null) then
      deleterow tf;
      set_forms form (change = 1);
    else
      message 'You can RowDelete only those rows that'
        + ' you have inserted (with RowInsert)'
        + ' and haven''t saved yet.'
        + ' To delete a record that''s already in'
        + ' the database, blank out its'
        + ' New PosCode field, or merge it with'
        + ' another record by setting its M field'
        + ' to "y" and changing its New PosCode'
        + ' to the position code of the other record.'
        with style = popup;
    endif;
  end
```

```
'RowInsert' (explanation = 'Open new row in table field') =
begin
  validrow tf; /* error if current row invalid */
  inquire_forms table '' (i = rowno);
  insertrow tf[i-1] (_state = 0);
end


'ReSelect' (explanation =
  'Refresh current row in tablefield from database') =
begin
  if (tf.old_position_code is null) then
    message 'You can ReSelect only those rows that'
      + ' were retrieved from the database.'
      with style = popup;
    resume;
  endif;

  /* Re-select the record from the database.
  ** If the database record has been deleted,
  ** delete the tablefield row.
  */
  repeated select
    :tf.position_code = m.position_code,
    :tf.position_title = m.position_title,
    :tf.position_description = m.position_description,
    :tf.merge = ' ',
    :tf.last_updated_at = m.last_updated_at,
    :tf.last_updated_by = m.last_updated_by
  from positions m
  where m.position_code = :tf.old_position_code;

  inquire_sql (row_count = rowcount);
  if (row_count <= 0) then
    deleterow tf;
    set_forms form (change = 1);
  endif;

  commit work; /* Release locks */
end
```

```
'MoreInfo' (explanation =
  'Access the job description information') =
begin


/* pass the current row of the tablefield to simple fields
on the description frame. The only field that is not
display-only field is the position_description field, so
this is passed using byref. Any changes made to this field
will be passed back to the calling frame. */



  i = callframe descriptions (
  samp_descriptions.position_code := iitf[].position_code,
  samp_descriptions.position_title := iitf[].position_title,
  samp_descriptions.last_updated_at := iitf[].last_updated_at,
  samp_descriptions.last_updated_by := iitf[].last_updated_by,
  position_description :=
      byref(iitf[].position_description));


  /* If the user has made changes on the description form, mark
  the position_description field changed
  A status = 1 means that the position_description field
  was changed in the descriptions frame */

  if i = 1 then
    SET_FORMS ROW '' iitf
      (CHANGE (position_description) = 1);
    SET_FORMS FORM (CHANGE = 1);
  endif;
end;
```

```
'Save' (activate = 1,
  explanation = 'Update database with current screen data'),
  key frskey8 (activate = 1) =
begin
  inquire_forms form (i = change);
  if (i = 1) then
    validate; /* validate all fields on form */
    message 'Saving changes . . .';

    /* process all rows */
    unloadtable tf (row_state = _state,
      row_number = _record)
    begin
          /* Check if the position_description field has been
          ** changed, since this value will be passed from the
          ** descriptions form using byref, the row_state will
          ** be unchanged */

      INQUIRE_FORMS ROW samp_positions iitf
      (i_descript = CHANGE(position_description));


      IF (row_state = 1 and i_descript = 0) THEN /* new */


        if (tf.position_code is not null) then
          repeated insert
          into positions(position_code, position_title
            position_description,
            last_updated_at, last_updated_by)
          values(tf.position_code, tf.position_title,
            tf.position_description,
            'now', user_name);

          if (check_io_err() != 0) then
            scroll tf to :row_number;
            row_number = 0;
            resume field tf;
          endif;
        endif;
```

```
elseif (row_state = 3 or i_descript = 1) then
   /* changed */

  /* The tablefield row has been marked as
  ** 'changed'. First, analyze which tables
  ** need records updated, deleted, or
  ** checked (for [non]existence). It's
  ** possible that no tables need to be
  ** changed: if the user changed a
  ** tablefield row and then did a ReSelect
  ** into it and made no further
  ** change, row_state will be 3.
  */
  verify_position = 'n';
  delete_position = 'n';
  update_position = 'n';
  update_employees = 'n';
  verify_no_employees = 'n';

  if (tf.position_code is null) then
    delete_position = 'y';
    verify_no_employees = 'y';
  elseif (tf.position_code !=
    tf.old_position_code) then
      if (uppercase(tf.merge) = 'Y') then
        verify_position = 'y';
        delete_position = 'y';
        update_employees = 'y';
      else
        update_position = 'y';
        update_employees = 'y';
      endif;
  else
    inquire_forms row samp_positions tf
      (i = change(position_title));
    if (i = 1 or i_descript = 1) then
      update_position = 'y';
    endif;
  endif;
```

```
/* If merging a position into a second
** position, verify that the second
** position exists. The Select puts a
** shared lock on the second position,
** so no other transaction can delete
** it while Employees is changed
** to the new position.
*/
if (verify_position = 'y') then

  repeated select i = 0 from positions
  where position_code = :tf.position_code;

  inquire_sql (row_count = rowcount);
  if (row_count <= 0) then
    rollback work;
    message 'The "Save" operation was'
      + ' not performed, because you'
      + ' are attempting to merge a'
      + ' record with a second'
      + ' record that does not'
      + ' exist. The cursor will be'
      + ' placed on the record.'
      with style = popup;
    scroll tf to :row_number;
    row_number = 0;
    resume field tf;
  endif;
endif;




/* Now, try to delete or update the Positions
** record represented by the tablefield row, as
** required. The Update or Delete places an
** exclusive lock on the Positions record.
** This prevents any Insert or Update to the
** Employee_histories table from adding a record
** by another user that specifies the
** position_code of the Positions record until
** the current user does a commit, since the Insert
** and Update in the Employees frame have a WHERE
** clause that refers to the position_code in the
** Positions table. This is an important
** consideration when deleting the Positions
** record or updating its key (position_code).
*/
if (delete_position = 'y') then


  /* Delete row using old key in WHERE clause. */
  repeated delete from positions
  where position_code = :tf.old_position_code
  and last_updated_at = :tf.last_updated_at;
```

```
elseif (update_position = 'y') then

  /* Update row using old key field in WHERE clause. */
  repeated update positions
  set position_code = :tf.position_code,
    position_title = :tf.position_title,
    position_description =
    :tf.position_description,
    last_updated_at = 'now',
    last_updated_by = :user_name
  where position_code = :tf.old_position_code
    and last_updated_at = :tf.last_updated_at;
endif;


  /* If trying to delete or update the positions
  ** record, see if successful.
  */
if (delete_position = 'y') or
  (update_position = 'y') then
    if (check_io_err() != 0) then
      scroll tf to :row_number;
      row_number = 0;
      resume field tf;
    endif;

    inquire_sql (row_count = rowcount);
    if (row_count <= 0) then
      rollback work;
      message 'The "Save" operation was not'
        + ' performed, because you are'
        + ' attempting to update or'
        + ' delete a record that has been'
        + ' updated or deleted by another'
        + ' user since you selected the'
        + ' record. The cursor will be'
        + ' placed on the record. You may'
        + ' wish to "ReSelect" the record'
        + ' to see what changes were made.'
        with style = popup;
      scroll tf to :row_number;
      row_number = 0;
      resume field tf;
    endif;
endif;
```

```
/* If deleting a position, verify that it doesn't
** occur in any records in Employee_histories.
*/
if (verify_no_employees = 'y') then

  repeated select i = 0 from employee_histories
  where position_code = :tf.old_position_code;


  inquire_sql (row_count = rowcount);
  if (row_count >= 0) then
    rollback work;
    message 'The "Save" operation was not'
      + ' performed, because you are'
      + ' attempting to delete a record'
      + ' whose position_code appears in'
      + ' records in the employee_histories'
      + ' table. The cursor will be'
      + ' placed on the record. You may'
      + ' wish to merge this Position'
      + ' record with another record by'
      + ' setting its M field to "y" and'
      + ' changing its New PosCode to the'
      + ' position code of the other record.'
      with style = popup;
    scroll tf to :row_number;
    row_number = 0;
    resume field tf;
  endif;
endif;
```

```
    /* If the user changes a position_code in a Positions
    ** record or merges one Positions record into another,
    ** change all occurrences of the position_code
    ** in the Employee_histories table.
    ** To adhere to proper optimistic locking protocol,
    ** first update the last_updated_at and last_updated_by
    ** columns in the corresponding master records
    ** in the Employees table. Thus, if a second user is
    ** currently working on one of the employees whose
    ** history is about to be updated by the current user,
    ** any attempt by the second user to save changes to
    ** the employee is flagged as an error.
    */


    if (update_employees = 'y') then

      repeated update employees m
      from employee_histories d
      set last_updated_at = 'now',
        last_updated_by = :user_name
      where m.ssn = d.ssn
      and d.position_code = :tf.old_position_code;

      if (check_io_err() != 0) then
        scroll tf to :row_number;
        row_number = 0;
        resume field tf;
      endif;

      repeated update employee_histories
      set position_code = :tf.position_code
      where position_code = :tf.old_position_code;

      if (check_io_err() != 0) then
        scroll tf to :row_number;
        row_number = 0;
        resume field tf;
      endif;
    endif;
  endif;
end; /* end of unloadtable */
```

```
    row_number = 0;

    commit work;

    if (check_io_err() != 0) then
      resume;
    endif;

    endloop;
  else
    message 'no changes to Save.' with style = popup;
  endif;
end /* end of 'Save' menuitem */


'ListChoices' (validate = 0, activate = 0,
  explanation = 'Show valid values for current field'),
  key frskey10 (validate = 0, activate = 0) =
begin
  i = callproc help_field;

  if (i > 0) then
     /* value was selected onto form */
    set_forms form (change = 1);
  endif;
end
```

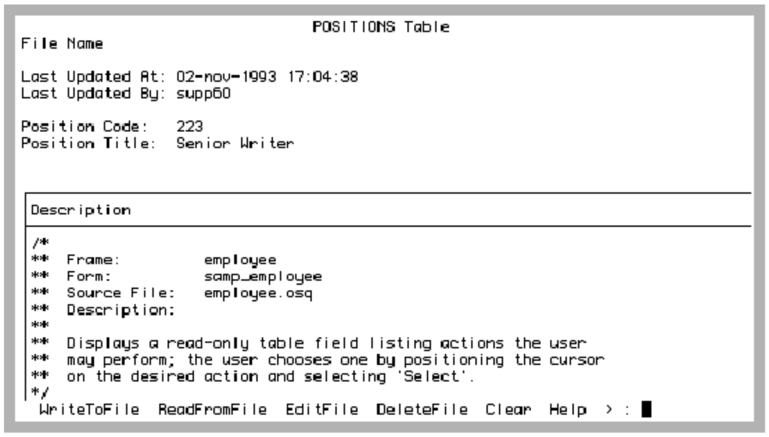
```
key frskey5 (explanation = 'scroll to top of table field') =
begin
  scroll tf to 1;
end

key frskey6 (explanation = 'scroll to bottom of table field') =
begin
  scroll tf to end;
end

key frskey7
(explanation = 'search table field for a specified value') =
begin
  i = callproc find_record;
end

'Help' (validate = 0, activate = 0,
  explanation = 'display help for this frame'),
  key frskey1 (validate = 0, activate = 0) =
begin
  if (menu_mode = 'a') then
    helpfile 'positions table'
      '/m/corgi/supp60/employee/positions.hla';
  else
    helpfile 'positions table'
      '/m/corgi/supp60/employee/positions.hlp';
  endif;
end

'End' (validate = 0, activate = 0,
  explanation =
  'Return from Update or Append mode to Query mode'),
  key frskey3 (validate = 0, activate = 0) =
begin
    inquire_forms form (i = change);
    if (i = 1) then
      char1 = callproc confirm (
        question = 'Do you wish to "End" without'
          + ' saving changes?',
        no = 'Cancel the "End" operation. (You can then'
          +'save your changes by selecting Save).',
        yes = '"End" without saving changes.'
      );
      if (char1 = 'n') then
        resume menu;
      endif;
    endif;
    endloop; /* exit submenu */
  end
  end; /* end of submenu in 'Go' menuitem */

  set_forms form (mode = 'query');   /* so user can enter
                                     ** another query */
  set_forms field samp_positions (mode(tf) = 'query');
  set_forms column samp_positions tf (displayonly(merge) = 1);
end


/*
** Local procedure: check_io_err
**
** Description:
**       Checks to see if the last database I/O statement
**       executed properly; does a ROLLBACK and puts out an
**       error message if not. If an error *has* occurred,
**       the DBMS has already issued its own error message.
**
** Returns:
**       The error number generated by the
**       database I/O statement (0 if no error).
*/
```

```
procedure check_io_err =
declare
  cursor_msg = varchar(80) not null,
  correct_msg = varchar(80) not null,
begin
  inquire_sql (error_no = dbmserror);
  if (error_no = 0) then
    return 0;
  endif;


  if (row_number <= 0) then
    cursor_msg = '';
  else
    cursor_msg = ' The cursor will be placed on the row'
      + ' where the error occurred.';
  endif;
  if (error_no = 4700) then /* deadlock (DBMS has already
                             ** done ROLLBACK) */
    correct_msg = '';  /* Nothing to fix on a
                       ** deadlock; just retry */
  else
    correct_msg = ' correct the error (if possible) and';
    rollback work;
  endif;


  message 'The "Save" operation was not performed, due to'
    + ' the error described in the previous message.'
    + cursor_msg
    + ' Please' + correct_msg + ' select "Save" again.'
    with style = popup;
  return error_no;
end
```

Descriptions Frame

The Descriptions frame lets users view the information held in the position_description field and import the information from or export it to an ASCII file on disk.

The form for the Descriptions frame is shown in the following figure:



```
                                  POSITIONS Table
File Name

Last Updated At: 02-nov-1993 17:04:38
Last Updated By: supp60

Position Code:   223
Position Title:  Senior Writer


 Description

 /*
 **  Frame:          employee
 **  Form:           samp_employee
 **  Source File:    employee.osq
 **  Description:
 **
 **  Displays a read-only table field listing actions the user
 **  may perform; the user chooses one by positioning the cursor
 **  on the desired action and selecting 'Select'.
 */
  WriteToFile  ReadFromFile  EditFile  DeleteFile  Clear  Help  > :
```

The position_code, position_title, last_updated_at, and last_updated_by fields are read-only fields passed to the form from the Positions frame. The position_description field is a table field "tbl" containing the contents of the position_description field. The changes made to this field are passed back to the Positions Frame. However, this does not change the field in the database. From the Positions Frame, you can select Save to update the field in the database.

Menu Operations

The Descriptions frame provides the following menu operations:

Menu Item	**Function**
WriteToFile	Writes the current contents of the description field to the file specified in the FileName field.
ReadFromFile	Reads the contents of the specified file to the descriptions field on the form. This does not change the field in the database.
EditFile	Invokes the system editor to edit the specified file.
DeleteFile	Deletes the specified file.
Clear	Clears all data from the screen
Help	Displays information about this frame

Menu Item	Function
End	Returns to the previous frame

The 4GL Source Code

The following is the 4GL source code for the Descriptions frame:

```
/*
** Frame: descriptions
** Form: samp_descriptions
** Source File: descriptions.osq
**
** Description
**
** Allows the user to import information from an ascii file on
** disk to fill the tbl tablefield, to export information
** from the tbl tablefield to an ascii file on disk, to delete
** an ascii file on disk and to invoke the systems editor to
** edit information stored in an ascii file on disk.
**
** Note, this frame will not save the data to the database. Once
** the user exits from this frame, the user will be returned to
** the positions frame and the position_description field on this
** frame will be returned to the positions frame, from whence it
** can be saved to the database.
**
** On entry to this frame, the simple variable
** position_description is unpacked into a tablefield of width 80
** and maximum length of 23 rows.
**


** On exit from this frame, the tablefield tbl is packed into the
** variable position_description and returned to the calling
** frame using byref.
**
** The file access functions will return a status of 0 for
** success and -1 for failure. No error will be returned to ABF.
**
** From the positions frame, call descriptions frame, passing
** values from all columns in the current row of positions.iitf
** table field to corresponding simple fields in descriptions
** form.
**
** A status = 0 is returned if no change is made to the
** position_description field, a status = 1 is returned
** otherwise.
*/
```

```
initialize (position_description = varchar(1900) not null ) =
declare


/* working variables */


  file_no = integer not null,    /* Holds handle for file
                                 ** manipulation */
  status = integer not null,     /* Holds 4gl statement status no */
  char1 = char(1) not null,      /* Holds answer to yes/no prompts */
  one_row = char(80) not null,   /* Holds each row while reading
                                 ** in from file */
  row_count = integer not null, /* Counter for number of rows
                                 ** read from ascii file */
  ncount = integer not null,     /* Holds the length of the position
                                 ** description field */
  system_param = varchar(80),    /* Holds the parameter string
                                 ** passed to invoke the editor */
  i = integer not null,          /* General purpose integer */
  changed = integer not null,    /* Tracks if user make changes to
                                 ** data */
  row_state = integer not null, /* Holds table field row state */


begin
  set_forms frs (activate(keys) = 1, activate(menuitem) = 1,
      validate(keys) = 1, validate(menuitem) = 1);

  changed = 0;


  /* load the tablefield from simple variable */


  ncount = length(ifnull(position_description,''));


  while ncount > 0 do


    loadtable tbl (description = left(position_description,80));


    ncount = length(position_description) - 80;


    position_description = right (position_description, ncount);


  endwhile;
end;
```

```
'WriteToFile' (explanation =
'Write Job Description information to file') =
Begin


  /* Check if the file exists, by opening a file in read mode.
  If the file does not exist, a status of -1 will return.*/


  status = callproc openfile (filename = :filename, filetype =
      'text', filemode = 'read', handle = byref(file_no));

  /* status = 0, implies that the file was opened successfully,
  so we want to warn users that they might be overwriting valuable
  information, and give the user the option of continuing or aborting
  the transaction */



  if (status = 0 ) then


    /* We had to open the file in read mode to check that
    it exists. We must close the file now since we have
    successfully opened it */


    status = callproc closefile (handle = file_no,
      disposition = 'keep');


    char1 = CALLPROC confirm (
     question = ifnull(filename,'')
       + ' already exists, and will be overwritten',
     no = 'Cancel the "WriteToFile" operation' +
      ' \ and prevent file from being overwritten.',
     yes = 'Continue and overwrite this file.'
     );
      IF (char1 = 'n') THEN
     RESUME;
   ENDIF;
  endif;
```

```
status = callproc openfile (filename = :filename, filetype =
  'text', filemode = 'create', handle = byref(file_no));


if status != 0 then
  /* ABF will not issue an error message to the user, so
  the code must issue the message */
  message 'File ' + ifnull(filename,'') +
   ' cannot be created'
  with style=popup;
  resume;
endif;


/* write data from the tablefield tbl to file,*/


unloadtable tbl (one_row = description)
begin


  if (tbl._state = 4) then
      endloop;
  endif;


  status = callproc writefile (handle = file_no,
      one_row);


  if status != 0 then
      endloop;
  endif;


end;


if status != 0 then
  /* ABF will not issue an error message to the user, so
  the code must issue the message */


  message 'Unable to write to file ' + ifnull(filename,'')
  with style=popup;
  resume;
endif;
```

```
  /* Close the file. This will flush the info to disk */

  status = callproc closefile (handle = file_no,
      disposition = 'keep');


  if status != 0 then
    /* ABF will not issue an error message to the user, so
    the code must issue the message */
    message 'File ' + ifnull(filename,'') + ' cannot be closed'
    with style=popup;
    resume;
  endif;


end; /* End of WriteToFile */


'ReadFromFile' (explanation =
'Read Job Description information from file') =
begin



  /* Open file in read mode */


  status = callproc OpenFile (filename = filename, filetype =
  'text', filemode = 'read', handle = byref(file_no));


  if status != 0 then
    /* ABF will not issue an error message to the user, so
    the code must issue the message */
    message 'Unable to open file ' + ifnull(filename,'')
    with style=popup;
    resume;
  endif;


  /* The tablefield tbl will hold 80x23 character rows.
  The variable one_row is 80 char in length and is used
  to hold one row of data, which is inserted into the tablefield
  tbl */
```

```
/* initialize the variables */


row_count = 0;


/* By reading in data from a file, the user is changing data,
use the variable changed to record that fact */


changed = 1;


/* Clear the tablefield */
clear field tbl;


while status = 0 and row_count < 23 do


  status = callproc ReadFile (  handle = file_no,
  byref(one_row));

  if status != 0 then
    endloop;
  endif;


  loadtable tbl (description = one_row);


  row_count = row_count + 1;


endwhile;
```

```
  /* close the file */


  status = callproc closefile (handle = file_no,
      disposition = 'keep');


  if status != 0 then
    /* ABF will not issue an error message to the user, so
    the code must issue the message */
    message 'Unable to close file ' + ifnull(filename,'')
    with style=popup;
    resume;
  endif;


end;




'EditFile'
(Explanation = 'Invoke the system editor to edit file') =
begin


  /* Invoke the system editor */


  system_param = '$ING_EDIT ' + ifnull(filename,'');
  call system :system_param;


end;


'DeleteFile'
(Explanation = 'Delete an ascii text file ') =
begin

  char1 = CALLPROC confirm (
 question = 'Are you sure you wish to delete ' +
      ifnull(filename,''),
 no = 'Cancel the "DeleteFile" operation' ,
 yes = 'Continue and delete this file.'
 );


  IF (char1 = 'n') THEN
     RESUME;
 ENDIF;
```

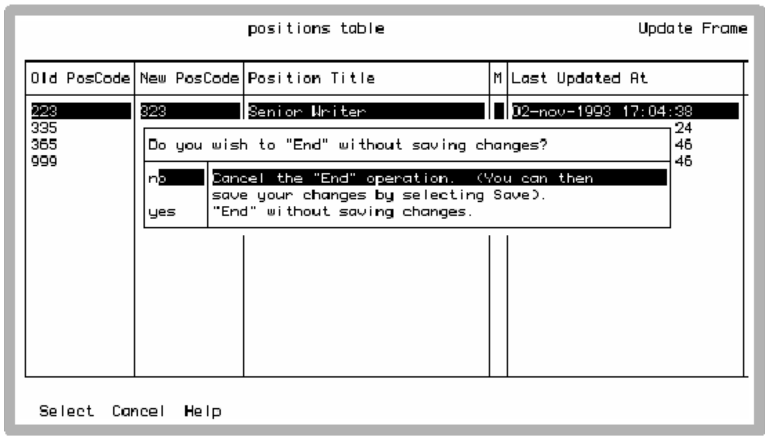
```
  /* A file can only be deleted when it is being closed. In
  order to close a file, it must be open, so first we open it */


  status = callproc OpenFile (filename = filename, filetype =
  'text', filemode = 'read', handle = byref(file_no));


  if status != 0 then
    /* ABF will not issue an error message to the user, so
    the code must issue the message */
    message 'File ' + ifnull(filename,'') +
       ' does not exists or is not accessible'
    with style = popup;
    resume;
  endif;



  /* close the file */


  status = callproc closefile (handle = file_no,
      disposition = 'delete');


  if status != 0 then
    /* ABF will not issue an error message to the user, so
    the code must issue the message */
    message 'Unable to delete file ' + ifnull(filename,'')
    with style=popup;
    resume;
  endif;
end;


'Clear' (validate = 0, activate = 0,
explanation = 'Clear Description') =
begin
  /* User is changing the contents of the field */
  changed = 1;


  clear field tbl;
end;



'Help' (validate = 0, activate = 0,
explanation = 'Display help for this frame'),
key frskey1 (validate = 0, activate = 0) =
begin
  helpfile 'descriptions frame'
  '/m/corgi/supp60/employee/descriptions.hlp';
end;
```

```
'End' (validate = 0, activate = 0,
explanation = 'Return to previous frame'),
key frskey3 (validate = 0, activate = 0) =
Begin
  /* Check if any change has been made to the position_description
  field. This could happen if the user tabs to the field (which
  will be ascertained using the _state variable ) or if the field
  is filled using the ReadFromFile function, in which case
  changed will be 1 */


    position_description = '';
    unloadtable tbl (row_state = _state)
    begin

        /* no need to include deleted rows */
      if (tbl._state = 4) then
        endloop;
      endif;


        /* Record the fact that data has changed */
      if (changed = 0 and row_state = 3) then
        changed = 1;
      endif;


      position_description = position_description +
              pad(tbl.description);
    end;


    /* if the field has been changed, check that it has data,
    because this is a required field */
    if changed = 1 and position_description = '' then
      message 'description is a required field' with
        style = popup;
      resume;
    endif;


      /* Return 1 if data has changed on this form,
      return 0 if no change has been made */
    return changed;


End;
```

Confirm Procedure

The Confirm procedure asks users to confirm that they want to do a potentially dangerous action, such as trying to exit from the Employees frame or the Positions frame without saving any changes that have been made.

The following figure shows an example of the pop-up form generated by the Confirm procedure (as it appears on the Positions frame):



```
                         positions table                          Update Frame

 Old PosCode|New PosCode|Position Title           |M|Last Updated At
 223        |823        |Senior Writer            | |02-nov-1993 17:04:38
 335        | Do you wish to "End" without saving changes?           24
 365        |                                                      46
 999        | no   | Cancel the "End" operation.  (You can then      46
            |      | save your changes by selecting Save).
            | yes  | "End" without saving changes.

  Select  Cancel  Help
```

The 4GL Source Code

The following is the 4GL source code for the Confirm procedure:

```
/*

/*
** Procedure: confirm
** Source File: confirm.osq
** Description:
**
** Prompts the user to confirm that a potentially dangerous
** action is to proceed. The prompting is done via the built-
** in frame look_up, which displays a popup like the following:
**
** +--------------------------------------------------------+
** |<question>                                              |
** +-------+------------------------------------------------+
** |no     |<no>                                            |
** |yes    |<yes>                                           |
** +-------+------------------------------------------------+
**** <question>, <no>, and <yes> are keyword parameters
** that you supply:
** <question> asks users if they really want to do the
** potentially dangerous action; <no> and <yes> provide
** explanations of the consequences of choosing 'no' or 'yes'.
**
** When the popup is displayed, the cursor is positioned on 'no'.
** The user can choose 'no' at this point by selecting
** 'Select', or choose 'yes' by scrolling down to it or
** hitting 'y' (which positions the cursor on the 'yes') and
** selecting 'Select'.
** The user can also select 'Cancel', which is equivalent to
** choosing 'no', no matter where the cursor is positioned.
**
```

```
** When you call this procedure, you can specify an
** explanation (<no> and/or <yes>) that occupies
** more than one line. To do this, embed
** a backslash character ('\') in the explanation at each
** point where a line break is to occur. The popup then
** displays blanks in the "choice" column of each continu-
** ation line. (The "choice" column is the left-hand column,
** which contains 'no' or 'yes' for non-continuation lines).
**
** The look_up frame allows the user to scroll into a
** continuation line (and "choose" it by selecting 'Select').
** If the user does choose a continuation line, this procedure
** acts as if the 'no' or 'yes' above it had been chosen.
**
** If an explanation is longer than 1000 bytes, it is truncated
** on the right. If a piece of an explanation (delimited by
** '\') is longer than 50 bytes (the size of the "explanation"
** attribute of the "choice_line" record type), it is .
** truncated on the right. If the question is longer than 57
** bytes, it too is truncated on the right.
**
** Returns:
**       'y' if the user chooses 'yes'.
**       'n' otherwise.
*/

procedure confirm (
  question = varchar(57) not null,
  no = varchar(1000) not null,
  yes = varchar(1000) not null,
) =
declare
  choice_array = array of choice_line,
  choice_index = integer not null,
  yes_index = integer not null,
  array_size = integer not null,
  fill_array = procedure returning integer not null,
begin
  yes_index = 1 + fill_array(starting_at = 1, explanation = no);
  array_size = fill_array(starting_at = yes_index,
    explanation = yes);

  choice_array[1].choice = 'no';
  choice_array[yes_index].choice = 'yes';

  choice_index = callframe look_up (
    ii_rows = array_size,
    ii_array = choice_array,
    ii_field1 = 'choice',
    ii_field2 = 'explanation',
    ii_title = question
  );
  /* If the user selected 'yes', look_up returns yes_index.
  ** If the user selected a blank choice below 'yes'
  ** (which represents a continuation of the explanation for
  ** 'yes'), look_up returns a value greater than yes_index.
  **
  ** If the user selected 'no', look_up returns 1.
  ** If the user selected a blank choice below 'no'
  ** (which represents a continuation of the explanation for
  ** 'no'), look_up returns a value greater than 1 but less
  ** than yes_index.
  **
  ** If the user cancelled the look_up operation,
  ** look_up returns 0 or a negative number.
  */
  if (choice_index >= yes_index) then
    return 'y';
  endif;
  return 'n';
end


/*
** Local procedure: fill_array
** Description:
**
** Fills rows of the array choice_array starting at the
** specified row. The "choice" attributes of the filled-in
** rows are set to ''. The "explanation" attributes of the
** filled-in rows are set to pieces of the specified
** explanation. (The pieces are separated by '\').
** As many rows are filled in as there are pieces in the
** specified explanation.
**
** Input parameters:
**    starting_at     the index of the first row to be filled in.
**    explanation     the explanation to be placed into the
**                    filled-in rows.
** Returns:
**    the index of the last row filled in.
*/
```

```
procedure fill_array (
  starting_at = integer not null,
  explanation = varchar(1000) not null,
) =
declare
  x = varchar(1000) not null,
  i = integer not null,
  b = integer not null,
 begin
  x = explanation;
  i = starting_at;
  b = locate(x, '\'); /* sets b to 1001 if no '\' found */
  while (b <= 1000) do
    choice_array[i].choice = '';
    choice_array[i].explanation = left(x, b - 1);
    x = right(x, length(x) - b);
    i = i + 1;
    b = locate(x, '\');
  /* sets b to 1001 if no '\' found */
  endwhile;
  choice_array[i].choice = '';
  choice_array[i].explanation = x;
  return i;
end
```

Appendix J: Notes for Users of QUEL

This section contains the following topics:

This appendix covers differences between QUEL syntax and 4GL. The differences appear by text chapter, and provide alternative examples written with QUEL syntax.

- In addition to the differences described in this appendix, note that some application development features are not supported in QUEL. QUEL does not support global variables, records, global application constants, or multiple sessions. These are supported for SQL only.
- For QUEL and 4GL reserved words, see Keywords (see page 1317).

In general, this appendix does not show examples in which the only difference between SQL and QUEL is QUEL's use of double quotes.

QUEL Notes for the Using 4GL Chapter

The sections below describe differences between QUEL and SQL as they apply to topics discussed in Using 4GL (see page 813).

Referencing Table Fields

Assign data to any or all of the columns of the table field with a database language retrieval statement.

The following statement reads data from the *projects* table into the data set, and then displays the data in the rows of the table field called *emptable*.

```
emptable := retrieve (projects.project,
 projects.hours);
```

To avoid ambiguity when the table field has the same name as the form, use the *form.tablefield* format:

```
emp.emp = retrieve
 (projects.project, projects.hours);
```

Data Types for Simple Local Variables and Hidden Columns

The format for declaring types in 4GL is the same as that used in create statements by the DBMS except that 4GL does not allow the with default or not default clauses.

- A *character value* is a sequence of characters bounded by *double* quotation marks in a QUEL implementation.
- *Nullable versions of each data type* are allowed in QUEL. By default, a type in QUEL is not nullable. You must use the with null clause to make it explicit. For example, to declare the type "nullable money," use the format money with null.

Simple Assignments

Place a constant value such as "Jones" in the character field *Name* by means of either of these assignment statements:

```
name := "Jones";
name = "Jones";
```

Query Assignment to a Form

In the following example, Empform is a form created with the VIFRED. The simple fields in the form correspond to Projname and Hours in the database. Ingres reads the data into the fields on the form.

```
empform = retrieve (projects.projname,
 projects.hours);
```

Note that it is not necessary that these fields have the same names as the names of database columns, but they must match in data type. If the names are different, the assignment statement must specify them as shown below:

```
empform = retrieve (task =
 projects.projname,
 time = projects.hours) ;
```

The retrieve statement is described in further detail below and in The Retrieve Statement (see page 1270) and Retrieve (QUEL) (see page 1296).

Query Assignment to a Table Field

A QUEL example of a retrieval into a table field is shown below:

```
partstbl := retrieve (parts.no, parts.name,
 parts.price);
```

Using Literals in Expressions

The literals in the following example are all expressions.

```
"J. J. Jones"
"Hendersonville"
"17-Aug-1998 10:00"
1209
7.77
```

In QUEL, double quotation marks are the standard means for representing string literals. Within a string literal, indicate a literal double quote by preceding it with a backslash character. For example:

```
"Quotation Mark \""
```

To specify a literal backslash character, use two backslashes. For example:

```
"backslash (\\)"
```

Specify date literals as strings and enclose them in double quotes (for example, "11/15/92"). Dates can be manipulated in date arithmetic operations.

Note: Beyond this point, this appendix does not show examples in which the only difference between SQL and QUEL is QUEL's use of double quotes.

Using Octal for Nonprintable Characters

In QUEL, specify nonprintable characters as a three-digit octal number preceded by a backslash (\). (Note that the hexadecimal notation described in Using 4GL (see page 813), is *not* used.) For example, the following command inserts the string "*XYZcarriage return*" into *col1* and a numeric into *col2* of *table1*:

```
append to table1 (col1 = "XYZ\015", col2 = 500);
```

The LIKE Operator and Pattern Matching

All the comparison operators used in 4GL require you to compare values for exact matching. 4GL has a pattern-matching capability that allows you to compare two strings and see whether they resemble each other in specified ways.

To do this, specify a pattern to determine what the compared string must look like, along with the LIKE pattern-matching operation. If the string you specify matches the pattern in the pattern string, the comparison evaluates to True. Use the LIKE operation in if and while statements. The syntax for the LIKE operation is as follows:

```
variable [NOT] LIKE pattern
```

The parameter *variable* refers to a character string column, and *pattern* is a character string literal or character string variable.

Pattern strings can include the following special characters:

- A question mark (?) is a "wild card" that matches any single character. For example, "Xa," "aa" and "/a" are all text strings that match the pattern string "?a."
- An asterisk (*) is a "wild card" that matches any string of characters, regardless of length. For example, the pattern string "Fred*" matches the strings "Fred," "Frederick," and "Fred S. Smith, Ph.D."
- Square brackets [ ] denote the beginning and end of a list of characters. Place brackets around the character in the string to be matched. For example, the pattern string "?[BC]C*" matches a string that has any character in the first position, a "B" or "C" in the second position, and a capital "C" in the third position.

The strings "ABC," "ACC," and "FCC Fairness Doctrine" are all valid instances of the pattern string "?[BC]C*" while "FDC Yellow #42" and "Access" are not.

Use the LIKE operator as any other comparison operator is used. For example, to test whether a text value ends in the letter "e," issue the following statement:

```
if name like "*e" then
 message "Found a name ending in e";
endif;
```

The following example tests the value of variable "emp_name" against the pattern "Fred*Smith" to see if it starts with "Fred" and ends with Smith.

```
if emp_name like "Fred*Smith" then
 message "Found a guy matching the pattern"
endif;
```

In QUEL, the pattern matching characters are always active in comparison clauses. The special meaning can be disabled by preceding it with a backslash (\). Note, however, that the pattern-matching characters operate like ordinary characters in *assignment* statements, as in the following:

```
jtitle := "**accountant**"
```

To retrieve the value requires the syntax shown in the following code fragment:

```
j.jtitle = "\*\*accountant\*\*"
```

Referencing Table Fields

Assign data to any or all of the columns of the table field with a database language retrieval statement. The following statement reads data from the *projects* table into the data set, and then displays the data in the rows of the table field called *emptable*.

```
emptable := retrieve (projects.project,
 projects.hours);
```

To avoid ambiguity when the table field has the same name as the form, use the *form*.*tablefield* format:

```
emp.emp = retrieve
 (projects.project, projects.hours);
```

QUEL Notes for the Writing 4GL Statements Chapter

The sections below describe differences between QUEL and SQL as they apply to topics discussed in Writing 4GL Statements (see page 849).

Using Unloadtable with Database Access

The QUEL application in the following example uses the _state constant and an unloadtable statement to update a database with mailing list information:

```
"Writelist" =

begin
  range of m is mailtable;
  unloadtable maillist (rowstat = _state)
  begin
    if rowstat = 1 then
      /* Add row to database */
      append mailtable
        (name = maillist.name,
        address = maillist.address,
        city = maillist.city,
        state = maillist.state);
    elseif rowstat = 3 then
      /* update the row in the database */
      replace m (name = maillist.name,
        address = maillist.address,
        city = maillist.city,
        state = maillist.state)
        where m.name = maillist.old;
    elseif rowstat = 4 then
      /* delete the row from the database */
      delete m where m.name = maillist.old;
    endif;
  end;
end
```

When this application is running, the user can choose Writelist from the menu to write changes back to the database. Ingres performs the appropriate database operation, depending on the value of the _state constant.

The key for the replace and delete statements is the table field's hidden column Old, which contains the value of the database column Name that was originally loaded from the database. The updated row must be identified by its previous contents.

Database Access Statements

In 4GL, you use these QUEL statements to query the database:

- retrieve (see page 1270)
- append (see page 1274)
- replace (see page 1274)
- delete (see page 1275)

These statements enable you to retrieve data, append rows to tables, update rows in tables, and delete information that is no longer needed.

The Retrieve Statement

To read data from database tables into forms and fields, use the retrieve statement. This statement takes values from the columns in the specified database table for all rows that satisfy the condition specified in a where clause. When you intend to get only a single row from the query statement (for example, in assigning values to the simple fields in a form), use the where clause to specify the row uniquely, as in this example:

```
deptform := retrieve (personnel.lname,
  personnel.fname)
  where personnel.empnum = 7;
```

This statement retrieves the name of employee number 7.

If you are retrieving many rows (as in a table-field assignment), use a **where** clause that specifies several rows, as shown in this example:

```
perstable := retrieve (personnel.lname,
  personnel.fname, personnel.salary)
  where personnel.salary >= 20000;
```

This statement retrieves a list of personnel who make at least $20,000.

Specify sorting criteria when the retrieval applies to several rows. Use the keywords sort by:

```
emptbl := retrieve (person =
  employee.ename, employee.dept)
  sort by person;
```

This example sorts a list of employees and their departments by employee name.

To specify sorting criteria and order, add ":a" (ascending—the default) or ":d" (descending). For example, to sort the above list in descending order by the name of the employee, specify the **sort** clause by:

```
sort by person:d;
```

Writing Different Types of Retrieve Statements

Several types of database query statements are possible in 4GL, allowing you to incorporate anything from a single item to an entire table on a form.

Using the Singleton Query

A *singleton query* gets values from a single row in the database table and places these values into simple fields on the form.

In the example below, the retrieve statement returns the single row that contains the part number in the *partnum* field.

```
editparts := retrieve (parts.name,
  parts.desc, parts.price)
  where parts.partno = partnum;
```

If more than one row in the database matches the conditions in the where clause, Ingres returns the first row that matches, as in this example:

```
empform := retrieve (emp.all)
  where dept = :dept;
```

Order is undefined unless you use the optional sort clause in the query. After a singleton query, rowcount is equal to at most 1.

Using the Table-Field Query

In a *table-field query*, Ingres retrieves a set of rows from the database into a table field on a form. The underlying data set of the table field receives all rows that match the query.

In this example, all the rows with an identification number greater than 100 are placed in the data set.

```
emptable := retrieve (emp.all)
  where idnum > 100;
```

Using the Attached Menu Query for Submenus

The *attached query* functions in QUEL as described for SQL. For example, the following query establishes a set of menu operations:

```
range of e is employee;
empform := retrieve(e.lname, e.age,
  e.empnum)
  where e.empnum < 10
begin
  "Msg" =
  begin
      message "Now within submenu";
      sleep 3;
  end
  "Increment" =
  begin
      age = age+1;
  end
  "NextRow" =
  begin
      next;
  end
  "End" =
  begin
      endloop;
  end
end
```

As Ingres displays each row of data, the user chooses whether to increment the age value, to go on to the next row, or to end. The endloop statement terminates the query.

Using the Master/Detail Query

The *Master/Detail query* functions the same way in QUEL as it does in SQL. The following example shows a QUEL Master/Detail database retrieval:

```
"Find" =
begin
  empinfo := retrieve (empname = staff.name,
    staff.title, staff.payrate)
  tasktable := retrieve (tasks.task,
    tasks.hours)
    where tasks.name = empname
  begin
    "Cancel" =
    begin
      /* Code to cancel the task */
      next;
    end
    "Update Hours" =
    begin
      /* Code to change hours on a task */
    end
    "Next" =
    begin
      next;
    end
    "End" =
    begin
      endloop;
    end
  end;
  clear field all;
end
```

In this example, the first query retrieves the master records into the *empinfo* form, and the second query retrieves the details for each master into the *tasktable* table field. The next statement steps through the master records. Each time a master row from the first query is displayed, the second query is run, based on the values from the *empname* column of the new master.

No semicolon is necessary between the two retrieve statements on the Master/Detail join, or between the second query statement and the begin statement (or starting brace) of the submenu.

Using a Retrieve Loop

The retrieve loop construction is analogous to the SQL select loop. Here is an example:

```
/* QUEL process employee update
 * transactions */

trans:= retrieve (h_trancode =
  transactions.trancode,
  h_empno = transactions.empno,
  h_salary = transactions.salary
begin
  /* Validate transaction */
  if (h_trancode = 1) then
    /* Salary update */
    repeat replace emp(salary = h_salary)
       where emp.empno = h_empno;
  else
    /* Process other transaction types */
  endif;
end;
```

Using Append to Add Data to Database Tables

The QUEL append statement is analogous to the SQL insert. The following QUEL append statement adds a row to the Personnel table, transferring the contents of simple fields *lname*, *fname*, and *empnum* to table columns *last*, *first*, and *empnum*, respectively.

```
append personnel (last = lname,
  first = fname,
  empnum = empnum);
```

Using Replace to Change a Database Row

The QUEL replace statement is analogous to the SQL update statement. The following QUEL replace statement replaces any row in the Personnel table that has a value in the *idnum* column equal to the value in the *idnum* field on the current form.

```
replace personnel (salary = sal,
  manager = manager)
  where personnel.idnum = idnum;
```

Deleting Database Rows

The QUEL delete statement below deletes the row that matches the employee number specified at run time:

```
delete personnel
  where personnel.empno = empno;
```

Committing the Transaction

In QUEL, transactions are individually opened and closed with each occurrence of a query statement (retrieve, append, replace and delete) unless you specifically establish a multi-query transaction (MQT). Use the following QUEL transaction control statements:

Begin transaction

Opens a transaction on the first occurrence of a query statement

End transaction

Commits the transaction and closes it

Abort

Terminates a transaction in progress, undoing the effects of all processed statements.

Savepoint

Declares a savepoint. Roll the transaction back to any of its savepoints prior to committing the transaction.

See the *QUEL Reference Guide* for detailed descriptions of these commands and a full description of transactions in QUEL.

Repeated Queries

An optimizing repeat function is available for the following database query statements: retrieve, append, replace, and delete.

The syntax for repeat queries varies from the standard data manipulation commands only in that the word repeat precedes the command:

```
repeat retrieve | append | replace | delete command
```

The optimization scheme is similar to that described for SQL queries.

The following example replaces information about an employee. Each time the query is run, a new employee number is specified, along with data currently on the form.

```
repeat replace employees (emptbl.all)
  where employees.empno = current_empno;
```

The next example deletes a sample test result. Each time the query is run, a new value for sample number and test result code is used.

```
repeat delete from results
  where results.sample_no = s_no
  and results.res_code = rescode;
```

The following example appends data on the form to the database table *customers*.

```
repeat append to customers (custform.all);
```

You cannot repeat the following types of queries:

- Retrieve into statements
- Queries that use the 4GL qualification function

  For example, the following is incorrect:

  ```
  /* incorrect */
      repeat delete employees
        where qualification (...);
  ```
- Queries that specify table and/or column names or the entire where clause in variables

Using All

The reserved word all enables you to transfer values between the database and all the displayed fields of a form in a single statement. This multiple assignment applies to simple fields or table-field columns but does not include hidden fields or columns.

This QUEL example uses the retrieve statement with all the simple fields on a form. It retrieves data from the columns in the Personnel table into all the simple fields of the form Deptform.

```
deptform = retrieve (personnel.all)
  where personnel.empnum = 6;
```

Using All with the Retrieve Statement

This QUEL example fetches data into all the columns of a table field. The form name is optional.

```
deptform.tfield = retrieve
  (personnel.all)
```

4GL expands the reserved word **all** based on the list of displayed simple fields or table-field columns in the current form or the specified table field.

For each simple field or table-field column, the table must have a column in the table with the same name and a compatible data type. The table can have unmatched columns (more table columns than form fields or table-field columns), but cannot have unmatched fields or table-field columns.

Using All with the Replace Statement

The multiple assignment is also possible with a QUEL **replace** statement. This statement updates a row in the database table Personnel from all the simple fields in Deptform.

```
replace personnel (deptform.all)
  where personnel.idnum = idnum;
```

Using All with the Append Statement

Use all with the QUEL append statement. The following statement adds a new row to the table called Part from all the visible columns in row 3 of the Partstbl table field.

```
append part (partstbl[3].all);
```

Using the Callframe Statement

To pass data from the database into a called frame with QUEL, use a retrieve statement to assign database values to a form or table field in the called frame, as shown below:

```
callframe newframe (newform = retrieve
  (projinfo.projnum, projinfo.projname)
  where projinfo.projnum = projno);
```

In this example, the current frame calls Newframe, passing values from the Projinfo table into a form named Newform. The where clause restricts the retrieval to a specific project number, which appears in the current frame's projno field.

Using the Inquire_ingres Statement

The following QUEL example uses rowcount to determine how many rows were found by a retrieve statement. It can also be used to see how many rows were added by an append statement or affected by a replace or delete statement:

```
partstbl := retrieve (parts.all);
inquire_ingres (rcount = rowcount);
if rcount < 1 then
  message "No records were found";
endif;
```

When used in conjunction with retrieve statements, the following rules apply:

- A singleton retrieve always causes inquire_ingres to return a value of 0 or 1 for rowcount.
- For attached queries, the value is the number of rows actually viewed by the user, as controlled by the next statement in the submenu.

This example uses errorno and errortext to check the results of a database operation:

```
delete employee
  where employee.empnum = empnum;
inquire_ingres (errno = errorno,
  txt = errortext);
if errno != 0 then
  message "Delete error " + varchar(errno) + " " + txt;
  sleep 3;
endif;
```

Using the Dbmsinfo Function

Use the 4GL dbmsinfo function with a retrieve statement to furnish information about the current Ingres process. Use this syntax:

```
formname := retrieve (fieldname = dbmsinfo
  (request name))
```

Formname is the name of the current form, *fieldname* is the name of a field into which the dbmsinfo results is retrieved, and *request name* is a quoted string corresponding to an Ingres literal constant containing the desired information.

The following example displays the version number of the current Ingres release in the Charfld field on the form.

```
thisform := retrieve (charfld =
  dbmsinfo("_version")) ;
```

This example retrieves and displays the user name of the current user.

```
thisform := retrieve (charfld =
  dbmsinfo("username")) ;
```

To compare the values of two dbmsinfo results, you must retrieve the results into fields in the form. The following example assigns the request names "dba" and "username" to the local variables *user1* and *user2* and compares them in an if then statement to determine if the current user is the DBA of the database. If so, the statements after the then clause, are processed:

```
thisform := retrieve (
  user1 = dbmsinfo("dba"),
  user2 = dbmsinfo("username"));
if user1 = user2 then
    perform actions
else
  message
    "You are not the DBA of this database";
  sleep 3;
endif;
```

Field Activations on Entry and Exit

The 4GL statements that accompany a field activation are executed whenever the user enters or leaves the field, as in this example:

```
field empnum =
begin
  if empnum = 0 or empnum is null then
    message
      "Please enter an employee number";
    sleep 3;
  else
    empform := retrieve (employee.fname,
      employee.lname)
      where employee.num = empnum;
    resume field "lname";
  endif;
end
```

When the user leaves the *empnum* field, Ingres checks the value in the field.

- *If the value in the field is zero,* it issues the error message and returns the cursor to the *empnum* field.
- *If the value in the field is not zero,* it retrieves the employee's name from the database and positions the cursor at the *lname* field.

QUEL Notes for the 4GL Statement Glossary Chapter

The sections below describe the QUEL statements, statement variants, and QUEL examples where they differ from those given in 4GL Statement Glossary (see page 935). Amend the statement descriptions as follows:

- Use retrieve, append, replace, delete, and range as described below in place of the SQL query statements (select/insert/update/delete) in 4GL Statement Glossary (see page 935).
- Use inquire_ingres and set_ingres instead of inquire_sql and set_sql. The statements are the same, except that the constants session, connection_name, and connection_target are not supported in QUEL.
- Use the call ingres|iquel|quel statement variants described below in place of the SQL versions in 4GL Statement Glossary (see page 935).
- For QUEL, statements are valid as they appear in 4GL Statement Glossary (see page 935). However, example sections for the statements that use queries or pattern matching are reproduced below to show the complete QUEL versions.
- QUEL does not support global variables, records, and global application constants. These are supported for SQL only.
- QUEL does not support connect and disconnect statements.

- QUEL does not support 'owner.tablename' syntax for objects.
- When quotes are indicated, QUEL statements require double quotes in place of the SQL single quotes.

Note: This appendix does not show examples in which the only difference between SQL and QUEL is QUEL's use of double quotes.

Append (QUEL)

Appends rows to a database table.

Syntax

```
[repeat] append [to] tablename (columnname =
  [tablefieldname[[integerexpr]].]fieldname
  {, columnname =
  [tablefieldname[[integerexpr]].]fieldname})
  [where qual]
```

tablename

Specifies the name of the table to which you are adding rows. This is a 4GL name.

columnname

Specifies the name of the database column to which you are assigning a particular value. This is a 4GL name. It need not be specified if it is identical to the name of the field on the form supplying the data.

tablefieldname

Specifies the name of the table field whose columns supply values for the append

integerexpr

Specifies an expression evaluating to an integer and representing the number of a row in a table field being used in the append statement

fieldname

Specifies the name of the simple field (or column in a table field) that contains the value to be inserted in *columnname*. When *fieldname* is a column in a table field, precede it with the name of the table field and a period, as shown in the syntax above.

qual

Specifies a logical expression that restricts the query to certain conditions. It cannot include 4GL names. A qualification clause is also allowed.

Description

The append statement allows you to add rows to a database table, using values from simple fields or the columns of a table field on a form. For maximum efficiency, use the repeat option to repeat an append statement.

A value in an append statement can come directly from the form, or it can be any legal expression, such as the result of a calculation involving a field in the form. In table fields, the value from the current row of the specified column is assumed. To append a value from a different row, specify the row number, enclosed in brackets or a pair of vertical bars, immediately following the table-field name. For example:

```
append part (partstbl[3].all);
```

Use append in conjunction with the unloadtable statement to add more than a single row at a time from a table field. For more information, see unloadtable Statement—Loop Through Rows and Execute Statements (see page 748).

- A where clause is optional. Note that the qualification function can be used in the where clause.
- The where clause can be stored in a variable. However, in this case, you cannot use the qualification function.
- Queries in which the qualification function is used are not suitable for use as repeated queries. The keyword repeat is ignored if you use it with such queries.

Use the keyword all with either a form name or a table-field name in place of the list of database column names. For example:

```
append personnel (deptform.all);
```

The database table must have a column that corresponds in both name and type to each displayed simple field in the form or to each column in the table field (the mapping ignores local variables).

Examples

Append the values in the current form to the *projects* table:

```
repeat append projects (name = projname,
  duedate = enddate);
```

Append the values in Empform to the Employee table, where fields and columns correspond:

```
append employee (empform.all);
```

Append the values in *empform* to the table specified in the field called *tablename* on the current form (fields and columns correspond):

```
tablename = "employee";
append :tablename (empform.all);
```

Append a computed value to the Personnel table:

```
append personnel (name = name,
  sal = salary * 1.1);
```

Call (QUEL)

The call statement has the QUEL variants listed below:

call ingres

Calls the QUEL Terminal Monitor

call iquel

Calls Interactive QUEL (IQUEL)

call quell

Calls the QUEL Terminal Monitors

These variants and QUEL examples are given below. See Call (see page 949) for other details regarding the call statement. For more information, see the *QUEL Reference Guide.*

Call Ingres

The statement call ingres calls the QUEL Terminal Monitor.

flags

Specifies the flags to be in effect. The *value* is a list of flags. Includes the following system level command line flags*:*

+a|-a
+d|-d
+s|-s

Separate items in the list with a blank or by pressing the Tab key.

Call Iquel

The statement call iquel calls Interactive QUEL.

flags

Specifies the flags to be in effect. The *value* is a list of flags. Includes the following system level command line flags*:*

+a|-a
+d|-d
+s|-s

Separate items in the list with a blank or by pressing the Tab key.

Call QUEL

The statement call quel calls the QUEL Terminal Monitor.

flags

Specifies the flags to be in effect. The *value* is a list of flags. Includes the following system level command line flags*:*

+a|-a
+d|-d
+s|-s

Separate items in the list with a blank or by pressing the Tab key.

Examples

Run QBF in the append mode, suppressing status messages, using the QBFName "expenses":

```
call qbf (qbfname = "expenses",
  flags = "-mappend -s");
```

Run a default report on Emptable in the column mode:

```
call report (name = "emptable",
  mode = "column");
```

Call report on a temporary table. The temporary table's name and contents are both based on user input. Note that in QUEL, embedded double quotes must always be dereferenced with a backslash (\), regardless of the operating system in use:

```
retrieve into :tbl (orders.all)
  where qualification (status = status,
    custno = custno);
call report (name = "orders",
  param = "table=\"" + :tbl + "\"")
```

The following example illustrates how to pass different data types in a **call report** statement:

- Use the following Report-Writer query specification:

```
.QUERY retrieve (callreptable.all)
where callreptable.db_int = $rw_int
and callreptable.db_char = '$rw_char'
and callreptable.db_date = '$rw_date'
```

- Include the following 4GL code in your application:

```
initialize (h_param = c80 not null) =
begin
end

"Callrep" =
begin

/* Construct a parameter string by      */
/* concatenating fields or variables with   */
/* text constants. Data comes from fields   */
/* "scr_int," "scr_char," and "scr_date,"   */
/* with data types INTEGER, CHAR, AND DATE,   */
/* respectively. As they are concatenated,   */
/* non-character fields are converted to   */
/* character type using the function VARCHAR. */

h_param = "rw_int=" + varchar(:scr_int)
  + "rw_char=\"" + :scr_char + "\""
  + " rw_date=\""
  + varchar(:scr_date) + "\"";
call report (report = 'callrep',
  param = :h_param);
end
```

Callframe (QUEL)

Transfers control from the current frame to another ABF frame.

The QUEL variants and examples for callframe are given below. For a complete description of the callframe statement, see Callframe (see page 965).

Passing a Query

4GL allows a query to be passed as a parameter to a form, utilizing the retrieve statement.

Use this syntax for the *parameterlist* to pass values from a database table:

```
calledformname := retrieve [unique]
  ([fieldname =] tablename.columnname
  {, [fieldname = ] tablename.columnname})
  [where qual]
  [sort [by] sortlist]
```

calledformname

Specifies the name of the form in the called frame

fieldname

Specifies the name of a simple field or table field in the called frame

tablename

Specifies the name of the database table from which values are being retrieved

columnname

Specifies the name of a column in the database table. If *columnname* is the same as *fieldname*, including *fieldname* is not necessary.

qual

Specifies a qualification clause or other logical expression that restricts the scope of the database query. The logical expression can not include 4GL names.

sortlist

Specifies a list of column names by which to sort the results of the retrieve statement

Only one query is allowed in the parameter list. Master/Detail queries cannot be passed as parameters.

A query passed as a parameter is similar to a query with a submenu. When the query is executed, it runs to completion, and the retrieved rows are read into a temporary file. Each next statement in the called frame causes the next row from the temporary file to be displayed.

Use the qualification function in a where clause, and enclose the entire where clause in a variable.

The section on the retrieve statement contains more detailed information on writing queries. However, some special rules apply when writing queries to be passed as parameters. To write the results of a query to a table field you must reference both the form and the table-field name, as in the example:

```
callframe newframe (newform.personnel :=
  retrieve ...) ;
```

Place the names of 4GL fields in the called frame on the left side of the target list, while names of components in the calling frame must appear on the right in the where clause. For example, consider the **callframe** statement:

```
callframe newframe (newform := retrieve
  empnum = projemps.empnum
  where projemps.projnum = :projnum) ;
```

Here Empnum is the name of a field in the called frame and therefore appears on the left side of the target list, while Projnum is a field in the calling frame and appears on the right in the where clause. 4GL does not generate errors until run time if fields in the left-hand side do not exist in the called frame.

To All Simple Fields from the Database

Use the syntax below for the parameterlist to retrieve values from the table into all the simple fields of the called form:

```
calledformname := retrieve [unique] (tablename.all)
  [where qual]
  [sort [by] sortlist]
```

To use the option all, you must be sure that each simple field in the called form corresponds by name to a column in the database table. There can be columns in the database table not corresponding to fields, but not vice versa.

To a Table Field from the Database

Use the syntax below for the *parameterlist* to pass values from a database table to a row (or rows) in a called table field:

```
calledformname.tablefieldname := retrieve [unique]
  ([tablefieldcolumn =] tablename.columnname
  {, [tablefieldcolumn =] tablename.columnname})
  [where qual]
  [sort [by] sortlist]
```

calledformname

Specifies the name of the form on the called frame

tablefieldname

Specifies the table field in the form on the called frame

tablefieldcolumn

Specifies the name of a table-field column on the called form to which database information is being assigned

The *tablename*, *columnname*, *qual*, and *sortlist* parameters have the same definitions as in the previous section.

If the database column has the same name as the table-field column, you need not specify *tablefieldcolumn* explicitly.

To Table-Field Columns from the Database

Use the following syntax for the *parameterlist* to retrieve values from the database table into all the columns of a table field:

```
calledformname.tablefieldname := retrieve
  [unique] (tablename.all)
  [where qual]
  [sort [by] sortlist]
```

All parameters have the same definitions as in the previous sections.

To use the option .all, you must make sure that each column in the table field corresponds by name and data type to a column in the table. The database table can contain columns that do not correspond to columns in the table field, but not vice versa.

Examples

Call Newframe and pass values into the Projnum and Name simple fields in Newform from the database:

```
callframe newframe (newform := retrieve
  (projinfo.projnum,
  name = projinfo.projname)
  where projinfo.projnum = projnum);
```

Call Newframe, passing values into the Projnum simple field and the Personnel table field of Newform. The example retrieves information from the Projemps table concerning all employees assigned to a particular project. The Personnel table field has at least the columns Empnum and Hours:

```
callframe newframe (newform.projnum :=
  projnum;
  newform.personnel := retrieve
    (projemps.empnum, hours = projemps.emphours)
    where projemps.projnum = projnum);
```

Call Newframe, placing a value in the status field upon returning to the calling frame:

```
status := callframe newframe;
```

Delete (QUEL)

Deletes rows from a database table.

Syntax

```
[repeat] delete tablename [where qual]
```

tablename

Specifies the name of the table from which rows are to be deleted. This is a 4GL name.

qual

Specifies a logical expression indicating which rows should be deleted from the table. It cannot contain 4GL names. The qualification function is allowed.

Description

The delete statement removes rows from a table based on the qualification in the where clause, if present. If no qualification is given, the statement deletes all the rows in the table.

- Use the qualification function in the where clause.
- Store the where clause in a variable. However, in this case, you cannot use the qualification statement.
- Queries in which the qualification function is used are not suitable for use as repeated queries. The keyword repeat is ignored if you use it with such queries.
- For maximum efficiency, repeat the delete statement, using the repeat option.

Deleting multiple rows from a table based on values from a table field normally involves the use of the unloadtable statement. See Unloadtable (see page 795).

Examples

Delete all rows in the personnel table containing the employee number (empno) displayed in the current form, using the repeat option:

```
repeat delete personnel
  where personnel.empno = empno;
```

Delete all rows in the Personnel table:

```
delete personnel;
```

Delete all rows in the table specified in the tablename field:

```
delete :tablename;
```

Delete rows from a table, where the table name and the where clause are expressed as variables:

```
whereclause = "empno = 12";
tablename = "personnel";
delete :tablename where :whereclause;
```

Message (QUEL)

Prints a message on the screen.

QUEL examples are given below. See Message (see page 1068) for a description of the message statement.

Examples

Display a message on the menu line until a database retrieval is completed:

```
message "Retrieving employee name";
empform := retrieve (emp.lname, emp.fname)
  where emp.empnum = empnum;
```

Display a pop-up style message on the number of rows retrieved:

```
empform := retrieve (emp.lname, emp.fname)
  where emp.empnum = empnum;
inquire_ingres (rcount = rowcount);
message "Retrieved" + varchar(rcount) +
  " rows" with style=popup;
```

Next (QUEL)

Displays the next row of retrieved data on the form.

QUEL examples are given below. See Next (see page 1074) for a description of the next statement.

Example (QUEL Only)

Retrieve employee information one row at a time into the simple fields of empform (this is an example of an attached query with a submenu):

```
empform := retrieve (employee.lname, employee.fname,
  employee.empnum)
  where employee.empnum <=100
  begin
    submenu statements here including:
    "NextRow" =
    begin
      next;
    end
  end;
```

Range (QUEL)

Declares a range variable for use in referring to a database table.

Syntax

```
range of rangevar is tablename {, rangevar is
  tablename}
```

rangevar

Specifies a synonym for a table or view. It cannot be a reserved word or the name of a field in the form. It is a 4GL name.

tablename

Specifies the name of the table or view for which the range variable substitutes. It is a 4GL name.

Description

The range statement declares range variables for use in subsequent query statements. A range variable acts as a synonym for the table name or view name and, once declared, can substitute for the table name or view name in any statement.

A range variable is generally declared in the initialize section of the frame that references the particular table. Up to 128 range variables can be in effect at any one time. In the case where a frame calls other frames that declare their own range variables, the initial frame's range variables can be lost if the total number of range variables exceeds 128. In this case, it can be preferable to define the range variables immediately before each query.

Examples

Declare range variables for the employee and dept tables:

```
range of e is employee, d is dept;
```

Declare a range variable for the table specified in the rangetable field:

```
range of e is :rangetable;
```

Redisplay (QUEL)

Refreshes frame and displays current values.

QUEL examples are given below. See Redisplay (see page 1084) for a description of the redisplay statement.

Example

Display a newly created part number before appending the values displayed in the form to the database table part:

```
"Add" =
begin
  editparts := retrieve (partno =
    max(part.partno) + 1);
  redisplay;
  sleep 2;
  append part (editparts.all);
end
```

Replace (QUEL)

Replaces values of columns in a database table.

Syntax

```
[repeat] replace tablename (columnname =
  [tablefieldname [[integerexpr]].]fieldname
  {, columnname =[tablefieldname[[integerexpr]].]
  fieldname})
   [where qual]
```

tablename

Specifies the name of the table in which columns are to be replaced; a 4GL name

columnname

Specifies the name of the column in the database table whose value is being updated; a 4GL name

tablefieldname

Specifies the table field containing the column from which the replacement value are taken

integerexpr

Specifies an expression that evaluates to an integer and that indicates the row number in the table field from which the value for an update is taken. If no integer expression appears in brackets, the current row is assumed.

fieldname

Specifies the name of a simple field (or of a column in a table field) that contains the value being used to update the column in the table. When *fieldname* is a column in a table field, it must be preceded by the name of the table field and a period, as shown.

qual

Specifies a logical expression that specifies which rows have column values replaced. It cannot contain 4GL names. The qualification function is allowed.

Description

The replace statement updates the values of columns in a database table, using values from simple fields or table-field columns on the current form. A value for updating a column can be any legal expression, such as a calculation involving a field in the form. For maximum efficiency, the replace statement can be repeated, using the repeat option.

- An optional where clause can be included to limit the rows in which the column values are changed.
- The qualification function can be used in the where clause.
- The where clause can be stored in a variable. However, in this case, you cannot use the qualification statement.
- Queries in which the qualification function is used are not suitable for use as repeated queries. The keyword repeat is ignored if you use it with such queries.

Updating a database table with values from many rows in a table field generally involves using the replace statement in conjunction with the unloadtable statement. The replace is located inside an unloadtable loop. See Unloadtable (see page 1156).

The replace statement can operate on several rows at a time. Therefore, use care in formulating the where clause so as to avoid unintentionally updating rows.

The reserved word all can be used with either a form name or a table field name. The table must contain a column that corresponds to each field or table-field column that is displayed on the form. The reserved word all is attached to the name of the form or table field, not to the name of the database table. In the case of the table field, all is used to indicate all the columns of the current row or of the row indicated by *integerexpr*, if there is one.

Examples

Replace values in the Projects table with values from the current form:

```
repeat replace projects (hours = hours,
  duedate = enddate)
  where projects.name = name;
```

Replace values in the Part table with values from the third row of the Partstbl table field in the current form:

```
replace part (partstbl[3].all)
  where part.partno = partstbl[3].partno;
```

Replace values in the Employee table with values from Empform (fields and columns correspond):

```
repeat replace employee (empform.all)
  where employee.lname = lname
  and employee.fname = fname;
```

Update the Personnel table with a computed value.

```
replace personnel (salary = sal*1.1)
  where personnel.empno = empno;
```

Update the Personnel table with a computed value; the table name and where clause are stored in variables:

```
tbl = "personnel";
whereclause = "personnel.empno" = char(empno);
replace :tbl (salary=sal*1.1)
  where :whereclause;
```

Retrieve (QUEL)

Retrieves rows from a database table.

Syntax

```
objectname := [repeat] retrieve [unique]
  ([fieldname = expression] | tablename.columnname
  {, [fieldname = expression] |
  tablename.columnname})
[where qual ]
[sort [by] sortlist] | [order [by] sortlist ]
[begin | {
  submenu
end | }]
```

objectname

Specifies the name of a form or of a table field in a form to which values are assigned. Where the table field name and form name are identical, specify *objectname* as *formname***.***tablefieldname.* Data types of *objectname* and *tablename***.***columnname* must be compatible.

fieldname

Specifies the name of a simple field on the form or of a column in the table field that receives the value from the specified database column

expression

Specifies any legal 4GL expression. The data types of the *expression* and *fieldname* must be compatible.

tablename.columnname

Specifies the name of the source table and column in the database table from which data is retrieved.

The name of the table cannot also be the name of a field on the form or a variable.

qual

Specifies a logical expression indicating conditions all rows retrieved must meet. It cannot include 4GL names. The qualification function is also allowed.

sortlist

Specifies a list of simple field names or table-field column names to serve as sort criteria. If more than one name appears in the list, 4GL sorts on the basis of the first, then on the basis of the second within the results of the first, and so on. To sort in ascending or descending order, append asc or desc to a field name after a colon (as in *namefield*:*desc).* You need not specify the default, asc.

submenu

Specifies a separate menu displayed for the duration of a retrieval. It lets the user exercise some options with respect to the data displayed, including displaying the next row of data. Separate the *submenu* from the rest of the retrieve statement by braces or by a begin and end pair.

Description

The retrieve statement has three variants:

Retrieve to a form or table field

Assigns values from rows in a database table to a form or a table field on a form. Combine this variant with an assignment statement in a variety of query formats to perform different kinds of assignments to forms or table fields. Attached queries (discussed below) can be nested, one within the other, to any level. The syntax for this variant is shown above.

Submenus can be used with attached queries only. They provide the user with the capability of displaying the next master row in the data set, or of performing an operation on the current row.

Retrieve into

Creates a new table and fills it with rows from other tables in the database that result from a query. The syntax for this variant is shown in its own section below.

Retrieve loop

Iterates through the rows of a query, assigning them to a form one row at a time to perform a sequence of 4GL statements for each row. The syntax for this variant is shown in its own section below.

For maximum performance, retrieve to a form object and retrieve loop can be repeated, using the reserved word repeat. You cannot repeat the retrieve into variant of this statement. Use the reserved word unique to eliminate duplicate rows.

Limitations to the retrieve statement:

- If the retrieve statement returns no rows, simple fields and table fields are not cleared. If it is necessary to clear these fields, issue a clear field statement before the retrieve.
- Only the table's owner and users with retrieve permission can retrieve from a table.

If a where clause is included, the retrieve statement returns values from the specified database table columns for all rows satisfying the where clause. The qualification function can be used in a where clause. Store the where clause in a variable. However, in this case, you cannot use the qualification statement.

Queries in which the qualification function is used are not suitable for use as repeated queries. The keyword repeat is ignored if you use it with such queries.

Use an optional sort or order clause to sort rows alphabetically or numerically, in ascending or descending order. The only difference between the two is that the sort clause removes duplicate rows, while the order clause does not.

These clauses are discussed in more detail in the following sections.

Query Targets

The target of this query can be of any Ingres data type, including a column or a row of a table field. The data types of the receiving field and the database column from which the data is retrieved must be compatible. If the field or table-field column has the same name as the database column from which the retrieve statement is fetching data, do not specify the field name.

Use the reserved word all in place of the target list in any retrieve statement to assign values to all the simple fields in a form or to all the visible columns of a table field. However, each simple field or table field column in the form must correspond to a database column with the same name and data type.

The database table can contain columns that do not match those in the form, but the form cannot contain unmatched simple fields or table field columns. Append the reserved word all to the tablename, as shown below:

```
form := retrieve (tablename.all);
```

The question mark (?) or asterisk (*) are wildcard characters. The question mark represents a single character while the asterisk represents any number of characters. For a list of comparison operators, see the section, Boolean Comparison Operators in Logical Expressions (see page 839).

The Where Clause

The optional where clause qualifies a database retrieval according to the specified logical (Boolean) expressions. The where clause is a Boolean combination of conditions (a QUEL predicate or an invocation of the qualification function).

Use a simple logical expression that compares a column in a database table to an expression, such as a constant value or a field in the current form. For example:

```
empform := retrieve
    (employee.name, employee.jobtitle)
  where employee.salary >= sallimit;
```

This statement retrieves into Empform the names and job titles of all employees whose salaries are greater than the value in the Sallimit field.

Place the qualification function in a where clause to allow the end user to include comparison operators (such as != or >) in the search condition at run time. The application user can type values, comparison operators, or wildcard characters into the visible simple fields in the current form specified by the qualification function.

The II_PATTERN_MATCH logical sets pattern matching to SQL or QUEL. Unless II_PATTERN_MATCH is set differently, you must use the wildcard characters for the query language of the application.

In QUEL, use the question mark (?) and the asterisk (*) as wildcards. The characters are defined during installation. The question mark (?) represents a single character, and the asterisk (*) represents any number of characters. For a list of comparison operators, see the section, Boolean Comparison Operators in Logical Expressions (see page 839).

The example below shows a query operation that utilizes the **qualification** function:

```
range of e is employee;
empform := retrieve (idnum = e.empnum,
  e.jobtitle, salary = e.empsal)
  where qualification (e.jobtitle = jobtitle,
    e.empsal = salary);
```

By entering the values "*Manager*" and "<30000" in the jobtitle and salary fields, respectively, the user retrieves information on all employees having a job title containing the word "Manager" and whose salaries are under $30,000.

The requirements for using the qualification function are:

- The current form must be in query mode.
- In the qualification function, only fields on a form are allowed. Local variables and hidden table-field columns are not allowed.
- Queries in which the qualification function is used are not suitable for use as repeated queries. The keyword repeat is ignored if you use it with such queries.

The Sort and Order Clauses

The sort clause removes duplicate rows from the retrieval and sorts the remaining rows on the basis of the specified simple fields of the current form (or columns of the table field).

The order clause is identical to the sort clause, except that it does not remove duplicate rows.

The full syntax for the sort and order clauses is as follows:

```
sort [by] fieldname[:sortorder]
  {, fieldname[:sortorder]} |
order [by] fieldname[:sortorder]
  {, fieldname[:sortorder]}
```

If more than one *fieldname* is specified in the sort or order clause, rows are sorted on the basis of the first *fieldname*, second *fieldname*, and so on, and within values of that field, on the basis of the specified *sortorder*. Specify a *sortorder* of asc or ascending (the default), or desc or descending.

The sort and order clauses require that you specify field names from the form, not column names from the database table. Consider the following examples:

```
empform := retrieve (employee.ename,
  employee.dept)
  sort by ename;
```

and

```
newform := retrieve (person=employee.ename,
  employee.dept)
  sort by person;
```

Both are correct. However, in the latter query it is incorrect to sort by Ename, because Ename is a column in the table and not a field in the form.

Retrieval into Complex Structures

Use the *retrieve to a form or table field* and *retrieve loop* variants of the retrieve statement with the assignment statement to perform selection into complex structures. The data types of the corresponding fields and columns must be compatible. Simple and complex assignment statements differ in the following ways:

- A simple assignment statement assigns a single value at a time.
- A complex assignment statement can assign one or more rows of values resulting from the retrieve operation to several simple fields or table-field columns.

4GL provides five kinds of complex assignment statements—*singleton queries*, *table-field queries*, *simple attached queries*, *Master/Detail attached queries*, and *retrieve loops*. When any one of these queries is executed, the query runs to completion, and the matching rows are stored in a temporary data set. The retrieve statement then accesses the data from the data set.

The first four of these assignment types are used with the statement variant, *retrieve to a form or table field*. They are discussed in more detail in Using Retrieve to a Form or Table Field (see page 1301). The fifth assignment type is discussed in Using Retrieve Loop (see page 1310).

Using Retrieve to a Form or Table Field

The variant *retrieve to a form or table field* is always associated with an assignment statement. The retrieve statement itself is a database expression, because it is equivalent to the values it retrieves. The syntax for *retrieve to a form or table field* is shown below:

```
objectname := [repeat] retrieve [unique]
  ([fieldname = expression] | tablename.columnname
  {, [fieldname = expression] |
  tablename.columnname})
[where qual ]
[sort [by] sortlist] | [order [by] sortlist ]
[begin | {
  submenu
end | }]
```

The retrieve statement makes up the right side of an assignment statement, while the name of the form or table field to which the retrieved values are assigned appears on the left side. If the retrieve returns no rows, then the previous contents of the field into which you are retrieving the data do not change.

There are four types of retrievals to a form or table field: the *singleton query*, the *table-field query*, the *simple attached query*, and the *Master/Detail query*. The latter two are attached queries, which use a form of the retrieve statement that includes a submenu. See Submenus.

Singleton Query

A *singleton query* retrieves and assigns a single row of values solely to the simple fields of the current form, without the use of a submenu. The query is stopped after the first row is retrieved, and only one row is displayed.

The following example of a singleton query retrieves a single row of values from the personnel table into a form named deptform:

```
deptform := retrieve (personnel.lname,
  personnel.fname)
  where personnel.empnum = 7;
```

Table-Field Query

A *table-field query* retrieves a set of forms and table fields from the database into a table field on a form, without the use of a submenu. The maximum number of rows that fit within the table field are displayed, and the remainder are buffered in the data set by the FRS. The user can view all the rows retrieved by scrolling through the table field.

The following is an example of a table-field query. It shows the retrieval of values from the projects table into a table field.

```
emptable = retrieve (projects.project,
  projects.hours)
  where projects.empnum = empnum;
```

Simple Attached Query and Submenu

A *simple attached query* retrieves and assigns several rows of data to the simple fields of a form. A *simple attached query* performs these functions:

- It *attaches* the as-yet-undisplayed rows to the form.
- It displays the first row in the simple fields of the form and uses a submenu to provide access to each of the subsequent rows.
- It displays a submenu of operations that can be carried out for each returned row.
- It provides submenu operations for displaying subsequent rows through a next statement. Submenus are discussed in more detail later in this section. See also Next (see page 1074).

An example of a simple attached query is shown below:

```
"Find" =
begin
  editparts := retrieve (partname =
    part.partname, partno = part.partno,
    descr = part.descr, cost = part.cost)
    where part.partno > 2666
  begin
    "Change" =
    begin
      /*code to replace the database entry */
      /*for the part with the new values */
      next;
    end
    "Delete" =
    begin
      /*code to delete the displayed part */
      /*from the database */
      next;
    end
    "IncreasePrice" =
    begin
      cost = cost * 1.1;
    end
    "Next" =
    begin
      next;
    end
    "End" =
    begin
      endloop;
    end
  end;
  message "Done with query";
  sleep 2;
  clear field all;
end
```

When the user chooses Find, the application displays on the form the first row retrieved, along with this submenu:

```
Change Delete IncreasePrice Next End
```

A single retrieve statement both starts the database retrieval and displays the new list of operations. 4GL combines these two actions within the statement because they are typically tied together in forms-based applications.

Master/Detail Query and Submenus

A *Master/Detail query* is a form of the attached query that retrieves data into the simple fields of a form (the master row) and also into a table field (the detail rows), then displays a submenu of operations that can be carried out for each master row retrieved. The first statement is the *parent* or *master* query; the second is the *child* or *detail* query.

Inclusion of a *submenu* offers the user a choice of operations that can be carried out for each master row retrieved, and provides the means for displaying the next master row. If the submenu is omitted, a single master row and all the details for that master are displayed at once; there is no way to see the next master row.

The Master/Detail query requires this syntax:

```
formname := [repeat] retrieve [unique]
    ([fieldname =] tablename.columnname
    {, [fieldname =] tablename.columnname})
[where qual ]
[sort [by] sortlist] | [order [by] sortlist ]
[formname.]tablefieldname :=
    [repeat] retrieve [unique]
    ([fieldname =] tablename.columnname
    {, [fieldname =] tablename.columnname})
[where qual ]
[sort [by] sortlist] | [order [by] sortlist ]
[begin | {
    submenu
end | }]
```

The only difference between this syntax and that of a simple attached query is that two queries are specified rather than one. Note that there is no semicolon between the two assignment statements. Only two assignment statements are allowed, and each must be to a form or table field.

An example of a Master/Detail query is as follows:

```
"Find" = begin addorders :=
  retrieve (orders.orderno,
    customer.custname)
    where qualification (orders.orderno =
    orderno, customer.custname = custname)
    and orders.custno = customer.custno
  partstbl := retrieve (part.partname,
    orderitems.qty, part.cost)
    where part.partno = orderitems.partno
    and orderitems.orderno = orderno
  begin
    "Cancel" = begin
      /* code to delete the order */
      next;
    end
    "Old" = begin
      /* code to change the order status */
      /* to old */
    end
    "Back" = begin
      /* code to change the order status */
      /* to back
    end
    "Next" = begin
      next;
    end

    "End" = begin
      endloop;
    end
  end;
  clear field all;
end
```

Each time a master row from the first query is displayed, the second query is run, based on the values from the new master. Once the master and the detail rows are displayed, 4GL prints this submenu on the window:

```
Cancel Old Back Next End
```

Choosing End returns the application user to the previous menu.

In a Master-Detail query, the where clause for the detail query is used to specify how the two queries are joined together. For instance, the detail query in the example shown above includes a reference to the *orderno* field, whose value was retrieved during the first query.

The value of the rowcount Ingres constant is always the number of master rows retrieved, whether or not a submenu is included. For a Master/Detail query with a submenu, the count is defined after the submenu ends and is set to the number of master records that were retrieved. The only exception to this is nested queries within submenus. See the section, Nesting Attached Queries.

Submenus

A *submenu* is a separate menu of choices that appears following the display of each master row in the retrieved data set for an attached query. Choosing an operation from the submenu causes the designated action to be performed, such as incrementing a salary field or displaying the next master row. Submenu operations always apply to the master row.

When a query with a submenu is executed, the query runs to completion, and a data set is created in a temporary file to hold the retrieved rows. A submenu appears at the bottom of the frame. Each next statement (associated with an item on the submenu) causes the next row in the data set to be displayed. If this is a Master/Detail query, the master query is performed as above, and another query runs to completion to retrieve the matching detail rows for each master row.

The syntax used to create a submenu is:

```
objectname := retrieve statement
[objectname := retrieve statement]
begin | {
  initialize = begin | {
    /* statements executed before first row */
    /* is displayed */
  end | }
  "menuitem1" = begin | {
    /* code for menuitem1 operation */
  end | }
  "menuitem2" =  begin | {
    /* code for menuitem2 operation */
  end | }
end | }
```

The submenu is separated from the rest of the retrieve statement by braces or by a begin and end pair. In the syntax of a Master/Detail query, the submenu appears after both assignment statements.

Submenus can contain all the same components as main menus. Generally, at least one of the menu operations should contain the 4GL next statement, which causes the next row retrieved to be displayed.

The user stays within the submenu until either (1) a next statement is executed after all the rows have been displayed, or (2) an endloop statement is executed. When either situation occurs, control returns to the statement immediately following the assignment statement that contained the submenu.

The following example of a simple attached query retrieves a series of row-typed values from the database and creates a submenu for the current form:

```
range of e is employee;
empform := retrieve (e.lname, e.age, e.empnum)
  where e.empnum <=10
begin
  initialize =
  begin
    /* Statements here are executed once */
    /* before the first row is displayed */
    message "Now within submenu";
    sleep 3;
  end
  field "lname" =
  begin
    message "You just entered value: " +
      lname;
    sleep 3;
    resume next;
  end
  "Increment" =
  begin
    age := age + 1;
  end

  "NextRow" =
  begin
    next;
  end
  "End" =
  begin
    endloop;
  end
end;
```

When the user chooses an operation in the main menu containing this assignment statement, the following submenu appears at the bottom of the window:

```
Increment NextRow End
```

The first row is retrieved in the three fields specified. Each time the user chooses the NextRow operation from the submenu, the next row is displayed. When there are no more rows, the message "No more rows" appears at the bottom of the frame, and the user exits from the current submenu. The main menu for the frame is again displayed.

If no rows in the database satisfy the query, the message "No rows retrieved" appears at the bottom of the frame, and control is transferred to the statements immediately following the assignment containing the submenu.

The submenu definition can contain any legal operations, including menu, key, and field activations. The first submenu operation definition can be an initialize block. Unlike the initialize block for a frame, the initialize block in an attached query cannot contain variable declarations. The initialize block for the attached query is executed only once, after the master row and matching details have been selected from the database and before their values have been displayed. Both 4GL and QUEL statements can appear in the initialize block.

The code in the submenu can contain any legal 4GL code. This includes query statements, such as replace and delete, as well as other assignment statements involving retrieves. A return or exit statement within the submenu has the same effect as a corresponding statement from the main menu of the frame. The resume and validate statements operate in the same manner as they do in the main menu; however, if they close the current operation, the submenu is still displayed.

While the submenu is running, the value of the rowcount FRS constant is undefined. Rowcount only has a value when the submenu ends. The single exception is when a query statement in the submenu code is executed, then rowcount is set for that query statement.

Note that when the assignment statement containing the submenu is executed, Application By Forms displays the current form in update mode. This mode can be changed in the initialize section of the submenu. When the end user exits from the submenu, the display mode in effect prior to the submenu again takes effect.

Nesting Attached Queries

4GL provides the capability to nest attached queries to any level. Each nested query is placed within the submenu of the higher-level query. Thus, the following syntax is valid:

```
formobject := database_expression
begin
  "menuitem" =
  begin
    formobject := database_expression
    begin
    {submenu}
    end
  end
end;
```

For nested queries with a submenu, the value of the rowcount FRS constant is undefined while the submenu is running. Rowcount only has a value when the submenu ends. The single exception when a query statement in the submenu code is executed; then the rowcount is set for that query statement.

Using Retrieve Into

The *retrieve into* statement is used to create a new table and fill it with rows from other tables in the database that result from a query. One particularly useful application of the retrieve into statement is the creation of a temporary table that can later be used by Report-Writer. Use the following syntax:

```
retrieve into tablename [unique]
  ([fieldname = expression] | tablename.columnname
  {, [fieldname = expression] |
  tablename.columnname})
[where qual ]
[sort [by] sortlist ] | [order [by] sortlist ]
[with clause]
```

The special functions and capabilities of the retrieve statement do not extend to the retrieve into variant. When used within 4GL, retrieve into stands alone as a separate statement. It never appears as a database expression in an assignment statement. It has these additional restrictions:

- The retrieve into statement cannot be repeated by the repeat reserved word.
- The unique keyword cannot follow the tablename specification; to retrieve unique rows, you must specify a sort by clause.

Using Retrieve Loop

The *retrieve loop* allows the program to perform a computation for each retrieved row, rather than to display results to the user, as in the *retrieve to a form or table field* statement.

A *retrieve loop* retrieves and assigns multiple rows of data, one row at a time, usually to the local variables in the current form. A series of specified operations is performed once on the values in each row retrieved. The syntax for *retrieve loop* is as follows:

```
[identifier:] [repeat] retrieve [unique]
  (fieldname [[ rowexpression].columnname] =
      expression | table.column)
[where qual ]
[sort [by] sortlist ] | [order [by] sortlist ]
[ begin | {
  statementlist
end | }]
```

This form of the statement executes commands in the *statementlist* once for each row returned by the query. The value retrieved from the right-hand side of the target list is assigned to the variables in the left-hand side of the target list.

- The left-hand side, if present, must be a legal 4GL variable reference. This variable can be a simple field, a local variable, a table-field column, or a hidden column.
- *If the left-hand side is not present,* then the right-hand side must be a database *table.column* reference, in which case the column's name must be the same as a simple field or local variable to which the value is assigned.

During execution of the loop, the value of inquire_ingres (rowcount) is not defined; if a query statement is executed within the loop, then rowcount is defined for that statement. After the retrieve loop, rowcount equals the number of rows through which the program looped.

Retrieve loops perform an implicit next operation each time the loop is executed. An explicit next statement within the body of the loop causes the next row of the data to be retrieved. Statement execution continues with the statement immediately following next; the next statement does not cause the loop to be executed again from the top. At the end, the loop is terminated. Use the endloop statement to terminate the loop early. For more information, see Endloop (see page 1008).

Examples

The first statements are examples of the first variant of retrieve, *retrieve to a form object*. The last statement is an example of the second variant, the *select loop*, which uses the begin and end statements.

Retrieve information about an employee based on the value in the Empnum field:

```
deptform := retrieve (personnel.last,
  personnel.first)
  where personnel.empnum = empnum;
```

Retrieve rows for all employees with an employee number greater than 100:

```
empform := retrieve (employee.lname,
  employee.fname, empnum = employee.number)
  where employee.number >100;
```

Retrieve into the Emptable table field all projects for the employee whose number is currently displayed in the Empnum simple field of the form:

```
emptable := retrieve (projects.project,
  projects.hours)
  where projects.empnum = empnum;
```

Retrieve and sort employee names matching the qualifications of the Empdept and Empsal fields:

```
empform := retrieve (emplname =
    employee.lname, empfname = employee.fname)
  where qualification (employee.dept =
    empdept, employee.sal = empsal)
  sort by emplname:a, empfname:a
begin
  /* submenu code goes here */
end;
```

Retrieve values from the Projects table into the Projform form, in which the simple fields and database columns correspond:

```
projform := retrieve (projects.all)
  where projects.empnum = empnum;
```

Retrieve user input into a temporary table whose name is also based on user input, and call a report on the temporary table. The report frame, Orderreport, has an associated blank form. Arguments such as the table name are passed as parameters to the call report statement:

```
retrieve into :tbl (orders.all)
  where qualification (status = status,
      custno = custno);
call report (name = orderreport, table = :tbl);
destroy :tbl;
```

Use a **retrieve** loop with **begin** and **end** statements to perform an operation on the row matching the **qualification** criteria:

```
getemp: retrieve (emp.all)
  where qualification (emp.lname = lname)
begin
  /* statements, including queries, */
  /* which are executed once for every */
  /* row retrieved */
end;
```

Unloadtable (QUEL)

Unloads the contents of the data set underlying a table field.

QUEL examples are given below. See Unloadtable (see page 1156) for a description of the unloadtable statement.

Examples (QUEL Only)

Use unloadtable with pattern matching to scroll the table field to the first row whose data starts with the characters the user specified. If the user enters "Smith" the pattern is set to "Smith*" and retrieves the first row beginning with the name Smith, Smithson, Smithburg, etc. The display then scrolls to the retrieved row.

```
pattern := name + "*";
unloadtable emp (row = _record)
begin
  if emp.name like pattern then
    scroll emp to row ;
  endif;
end;
```

The example below updates a mailing list on the basis of the _state of each data set row. The hidden table-field column called *maillist.old* is the key to the updating, because it contains the original value of the Name field when the row was originally loaded from the database into the data set:

```
"Writelist" =
begin
  unloadtable maillist (rowstat = _state)
  begin
    if rowstat = 1 then
      /* Add row to database */
      repeat append mailtable
        (name = maillist.name,
        address = maillist.address,
        city = maillist.city,
        state = maillist.state);


    elseif rowstat = 3 then
      /* Update the row in the database */
      repeat replace mailtable
        (name = maillist.name,
        address = maillist.address,
        city = maillist.city,
        state = maillist.state)
        where mailtable.name =
          maillist.old;
    elseif rowstat = 4 then
      /* Delete the row from the */
      /* database */
      repeat delete mailtable
        where mailtable.name =
          maillist.old;
    endif;
  end;
```

While (QUEL)

Repeats a series of statements while a specified condition is true.

QUEL examples are given below. See While (see page 1169) for a description of the while statement.

Examples (QUEL Only)

Use the while statement to handle deadlock:

```
succeed := 0;     /* Set to 1 when and if */
             /* append is successful */
tries := 1;
ingerr := 0;

A:   while succeed = 0 and tries <= 10 do
    ...append statement to try to append
    to database...
    inquire_ingres (ingerr = "errorno");
    if ingerr = 0 then
      /* Append completed successfully */
      succeed := 1;
    elseif ingerr = 4700 then
      /* Error due to deadlock - try again */
      tries := tries + 1;
    else
      endloop A;
    endif;
  endwhile;
if succeed = 1 then
  message "Append completed successfully";
  sleep 3;
else
  message "Append failed";
  sleep 3;
endif;
```

QUEL Notes for the Returning to the Top Frame Appendix

The following notes describe differences between 4GL and QUEL in Returning to the Top Frame (see page 1337). The general discussion in that appendix applies to QUEL.

Embedded Query Language Considerations

This section points out the special considerations for EQUEL in returning to the top frame.

You should not return immediately from inside a display loop, a retrieve loop, or from a routine that has cursors open. Exit each of these cleanly, as follows, with the appropriate EQUEL statements before returning to the calling frame:

- Break out of a display loop with the breakdisplay or enddisplay statement.
- Exit from a retrieve loop with the endretrieve statement.
- Close open cursors with the close statement.

For example, the following section shows a program fragment that illustrates returning a status of "11" from inside a display loop.

Example (EQUEL Only)

```
## ACTIVATE MENUITEM "Top";
## {
##    status = 11 ;
##    BREAKDISPLAY    /* drop out of display loop */
## /* Note: don't want to do
## * 'return (11);' here because we're
## * inside a display loop */
## }
## finalize()
    return ( status ) ;
```

QUEL Notes for the Runtime Error Processing Appendix

The comments in this appendix apply equally to EQUEL, except as noted below.

Effect on EQUEL Procedures

In ABF/QUEL applications, the user-defined Ingres error handler affects all frames, 4GL procedures, and EQUEL procedures.

Appendix K: Keywords

This section contains the following topics:

This appendix lists Ingres keywords and ISO standard keywords. These lists enable you to avoid assigning object names that conflict with reserved words.

Ingres Keywords

The following table lists Ingres keywords and indicates the contexts in which they are reserved.

Note: The keywords in this list do not necessarily correspond to supported features. Some words are reserved for future or internal use, and some words are reserved to provide backward compatibility with older features.

In the following table, the column headings have the following meanings:

ISQL – Interactive SQL

These keywords are reserved by the DBMS.

ESQL – Embedded SQL

These keywords are reserved by the SQL preprocessors.

IQUEL – Interactive QUEL

These keywords are reserved by the DBMS.

EQUEL – Embedded QUEL

These keywords are reserved by the QUEL preprocessors.

4GL – Ingres 4GL

These keywords are reserved in the context of SQL or QUEL in 4GL routines.

Note: The ESQL and EQUEL preprocessors also reserve forms statements. Forms statements are described in the Embedded Forms Programming part of this guide.

Single keywords are listed in the following table:

Keyword	**SQL**			**QUEL**		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
abort	*	*	*	*	*	*
activate		*			*	
add	*	*	*			
addform		*			*	
after			*			*
all	*	*		*	*	
alter	*		*			
and	*	*		*	*	
any	*	*	*	*	*	
append				*	*	*
array			*			
as	*	*		*	*	*
asc	*		*			
at	*	*	*	*	*	*
authorization	*	*				
avg	*	*	*	*	*	
avgu		*		*	*	
before			*			*
begin	*	*	*	*		*
between	*	*	*			
breakdisplay		*			*	
by	*	*		*	*	*
byref	*	*	*			*
call		*	*		*	*
callframe			*			*
callproc	*		*			*
cascade	*	*				

Keyword	**SQL**			**QUEL**		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
case	*	*				
check	*	*	*			
clear		*	*		*	*
clearrow		*	*		*	*
close	*	*		*		
column		*			*	
command		*			*	
comment	*		*			
commit	*	*				
connect		*				
constraint	*	*	*			
continue	*	*				
copy	*	*	*	*	*	*
count	*	*	*	*	*	
countu		*		*	*	
create	*	*	*	*	*	*
current	*	*				
current_user	*	*	*			
cursor	*	*				
datahandler		*				
dbms_password		*				
declare	*	*	*			*
default	*	*	*			*
define	*			*		*
delete	*	*	*	*	*	*
deleterow		*	*		*	*
desc			*			
describe	*	*				
descriptor		*				

Keyword	SQL			QUEL		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
destroy				*	*	*
direct			*			*
disable	*		*			
disconnect		*				
display		*	*		*	*
distinct	*	*	*			
distribute				*		
do	*		*			*
down		*			*	
drop	*	*	*			
else	*	*	*			*
elseif	*		*			*
enable	*		*			
end	*	*	*	*	*	*
end-exec		*				
enddata		*			*	
enddisplay		*			*	
endforms		*			*	
endif	*		*			*
endloop	*	*	*		*	*
endretrieve					*	
endselect		*				
endwhile	*		*			*
escape	*	*				
exclude				*		
excluding	*	*		*		
execute	*	*		*		
exists	*	*	*			
exit			*		*	*

Keyword	**SQL**			**QUEL**		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
fetch	*	*				
field		*			*	
finalize		*			*	
for	*	*	*	*	*	
foreign	*	*	*			
formdata		*			*	
forminit		*			*	
forms		*			*	
from	*	*	*	*	*	*
full	*	*	*			
get			*			
getform		*			*	
getoper		*			*	
getrow		*			*	
global	*	*	*			
goto		*				
grant	*	*	*			
group	*	*	*			
having	*	*	*			
help		*		*	*	
help_forms			*			*
help_frs		*			*	
helpfile		*	*		*	*
identified		*	*			
if	*	*	*			*
iimessage		*			*	
iiprintf		*			*	
iiprompt		*			*	
iistatement					*	

Keyword	**SQL**			**QUEL**		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
immediate	*	*	*			*
import	*					
in	*	*	*	*	*	
include		*		*		
index	*	*	*	*	*	*
indicator		*				
ingres					*	
initial_user	*	*				
initialize		*	*		*	*
inittable		*	*		*	*
inner	*	*	*			
inquire_equel					*	
inquire_forms			*			*
inquire_frs		*			*	
inquire_ingres		*	*		*	*
inquire_sql		*	*			
insert	*	*	*			
insertrow		*	*		*	*
integrity	*	*		*		*
into	*	*	*	*	*	*
is	*	*	*	*	*	*
isolation	*					
join	*	*				
key	*	*	*			*
left	*	*	*			
level	*					
like	*	*				
loadtable		*	*		*	*
local	*					

Keyword	**SQL**			**QUEL**		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
max	*	*	*	*	*	
menuitem		*			*	
message	*	*	*		*	*
min	*	*	*	*	*	
mode			*			*
modify	*	*	*	*	*	*
module	*					
move				*		
natural	*	*				
next		*			*	
noecho			*			*
not	*	*		*	*	
notrim		*			*	
null	*	*	*		*	*
of	*	*	*	*	*	*
off			*			*
on	*	*		*	*	*
only	*			*		*
open	*	*		*		
option	*					
or	*	*		*	*	
order	*	*	*	*	*	*
out		*				
outer	*					
param					*	
permit	*	*		*		*
prepare	*	*				
preserve	*	*				
primary	*	*	*			

Keyword	**SQL**			**QUEL**		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
print		*		*	*	
printscreen		*	*		*	*
privileges	*					
procedure	*	*	*			*
prompt		*	*		*	*
public	*	*				
putform		*			*	
putoper		*			*	
putrow		*			*	
qualification			*			*
raise	*		*			
range				*	*	*
read	*	*		*		
redisplay		*	*		*	*
references	*	*	*			
referencing	*		*			
register	*	*	*		*	*
relocate	*	*	*	*	*	*
remove	*	*	*		*	*
rename				*		
repeat	*	*	*		*	*
repeated		*	*			
repeatable	*					
replace				*	*	*
replicate				*		
restrict	*	*				
resume		*	*		*	*
retrieve				*	*	*
return	*		*			*

Keyword	**SQL**			**QUEL**		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
revoke	*	*	*			
right	*	*	*			
rollback	*	*	*			
rows	*	*				
run			*			*
save	*	*	*	*	*	*
savepoint	*	*	*	*	*	*
schema	*	*				
screen		*	*		*	*
scroll		*	*		*	*
scrolldown		*			*	
scrollup		*			*	
section		*				
select	*	*	*			
serializeable	*	*				
session	*	*				
session_user	*	*				
set	*	*	*	*	*	*
set_4gl			*			*
set_equel					*	
set_forms			*			*
set_frs		*			*	
set_ingres		*	*		*	*
set_sql		*	*			
sleep		*	*		*	*
some	*	*	*			
sort				*	*	*
sql	*					
stop		*				

Keyword	SQL			QUEL		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
submenu		*			*	
sum	*	*	*	*	*	
sumu		*		*	*	
system	*		*			*
system_maintained	*	*				
system_user		*				
table	*	*				
tabledata		*			*	
temporary	*	*				
then	*	*	*			*
to	*	*		*	*	*
type			*			
uncommitted	*	*				
union	*	*	*			
unique	*	*	*	*	*	*
unloadtable		*	*		*	*
until	*	*	*	*	*	*
up		*			*	
update	*	*	*	*		
user	*	*	*			
using	*	*				
validate		*	*		*	*
validrow		*	*		*	*
values	*	*	*			
view	*	*		*		*
when	*	*				
whenever		*				
where	*	*	*	*	*	*
while	*					*

Keyword	**SQL**			**QUEL**		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
with	*	*	*	*	*	*
work	*		*			
write	*			*		

Double keywords are listed in the following table:

Double Keyword	**SQL**			**QUEL**		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
after field			*			*
alter group	*	*	*			
alter location	*	*	*			
alter profile	*	*				
alter role	*	*	*			
alter security_audit	*	*	*			
alter table	*	*	*			
alter user	*	*	*			
array of			*			
before field			*			*
begin declare		*				
begin transaction	*	*	*	*	*	*
by user	*		*			
call on			*			
call procedure			*			
class of			*			
close cursor				*	*	
comment on	*	*	*			
connect to			*			
copy table			*			
create dbevent	*	*	*			

Double Keyword	SQL			QUEL		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
create group	*		*			
create integrity	*		*			
create link	*	*				
create location	*	*	*			
create permit	*		*			
create procedure			*			
create profile	*	*				
create role	*	*	*			
create rule	*	*	*			
create security_alarm	*	*	*			
create synonym	*	*	*			
create user	*	*	*			
create view	*		*			
current installation			*			
define cursor				*		
declare cursor					*	
define integrity				*	*	*
define link					*	
define location				*		
define permit				*	*	*
define qry				*		*
define query				*		
define view				*	*	*
delete cursor					*	
destroy integrity				*	*	*
destroy link					*	
destroy permit				*	*	*
destroy table					*	
destroy view						*

Double Keyword	SQL			QUEL		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
direct connect		*	*		*	*
direct disconnect		*	*		*	*
direct execute		*				*
disable security_audit	*	*	*			
disconnect current			*			
display submenu			*			*
drop dbevent	*	*	*			
drop group	*		*			
drop integrity	*		*			
drop link	*	*	*			
drop location	*	*	*			
drop permit	*		*			
drop procedure			*			
drop profile	*	*	*			
drop role	*	*	*			
drop rule	*	*	*			
drop security_alarm	*	*	*			
drop synonym	*	*	*			
drop user	*	*	*			
drop view	*		*			
each row		*				
each statement		*				
enable security_audit	*	*	*			
end transaction	*	*	*	*	*	*
exec sql						
execute immediate			*			
execute on			*			
execute procedure			*			
foreign key			*			

Double Keyword	**SQL**			**QUEL**		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
for deferred	*			*		
for direct	*			*		
for readonly	*			*		
for retrieve				*		
for update				*		
from group	*		*			
from role	*		*			
from user	*		*			
full join	*		*			
full outer	*					
get data		*				
get dbevent		*	*			
global temporary			*			
help comment	*					
help integrity		*			*	
help permit		*			*	
help table	*					
help view		*			*	
identified by			*			
inner join	*		*			
is null				*		
left join	*		*			
left outer	*					
modify table			*			
not like	*		*			*
not null				*		
on commit	*	*	*			
on current	*					
on database	*	*	*			

Double Keyword	SQL			QUEL		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
on dbevent	*		*			
on location	*		*			
on procedure	*					
only where				*		
open cursor				*	*	
order by				*		
primary key			*			
procedure returning			*			*
put data		*				
raise dbevent	*	*	*			
raise error	*					
register dbevent	*	*	*			
register table						*
register view			*			*
remote system_password		*				
remote system_user		*				
remove dbevent	*	*	*			
remove table						*
remove view			*			*
replace cursor				*	*	*
resume entry			*			*
resume menu			*			*
resume next			*			*
retrieve cursor				*	*	
right join	*		*			
right outer	*					
run submenu			*			*
session group			*			
session role			*			

Double Keyword	SQL			QUEL		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
session user			*			
set aggregate	*			*		
set autocommit	*			*		
set connection		*				
set cpufactor	*			*		
set date_format	*			*		
set ddl_concurrency	*					
set decimal	*			*		
set flatten	*			*		
set io_trace	*			*		
set jcpufactor				*		
set joinop	*			*		
set journaling	*			*		
set lock_trace	*			*		
set lockmode	*			*		
set logdbevents	*					
set log_trace	*			*		
set logging	*			*		
set maxconnect	*					
set maxcost	*			*		
set maxcpu	*			*		
set maxidle	*					
set maxio	*			*		
set maxpage	*			*		
set maxquery	*			*		
set maxrow	*			*		
set money_format	*			*		
set money_prec	*			*		
set noflatten	*			*		

Double Keyword	**SQL**			**QUEL**		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
set noio_trace	*			*		
set nojoinop	*			*		
set nojournaling	*			*		
set nolock_trace	*			*		
set nologdbevents	*					
set nolog_trace	*			*		
set nologging	*			*		
set nomaxconnect	*					
set nomaxcost	*			*		
set nomaxcpu	*			*		
set nomaxidle	*					
set nomaxio	*			*		
set nomaxpage	*			*		
set nomaxquery	*			*		
set nomaxrow	*			*		
set nooptimizeonly	*			*		
set noprintdbevents	*					
set noprintqry	*			*		
set noprintrules	*					
set noqep	*			*		
set norules	*					
set nosql				*		
set nostatistics	*			*		
set notrace	*			*		
set optimizeonly	*			*		
set printdbevents	*					
set printqry	*			*		
set printrules	*					
set qep	*			*		

Double Keyword	SQL			QUEL		
Reserved in:	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
set random_seed	*			*		
set result_structure	*			*		
set ret_into				*		
set role	*					
set rules	*					
set session	*			*		
set sql				*		
set statistics	*			*		
set trace	*			*		
set transaction	*					
set update_rowcount	*			*		
set work	*					
system user			*			
to group	*		*			
to role	*		*			
to user	*	*				
user authorization			*			
with null				*		
with short_remark	*					

ISO SQL

The following keywords are ISO standard keywords that are not currently reserved in Ingres SQL or Ingres Embedded SQL. Treat these as reserved words to ensure compatibility with other implementations of SQL.

absolute	except	output
action	exception	overlaps
allocate	exec	pad
are	external	partial

assertion
bit
bit_length
both
cascaded
cast
catalog
char
character
char_length
character_length
coalesce
collate
collation
connection
constraints
convert
corresponding
cross
current_date
current_time
current_timestamp
date
day
deallocate
dec
decimal
deferrable
deferred
desc
diagnostics
domain
double
extract
false
first
float
found
get
go
hour
identity
initially
input
insensitive
int
integer
intersects
interval
language
last
leading
level
lower
match
minute
module
month
names
national
nchar
no
nullif
numeric
octet_length
option
position
precision
prior
privileges
real
relative
second
size
smallint
space
sql
sqlcode
sqlerror
sqlstate
substring
then
time
timestamp
timezone_hour
timezone_minute
trailing
transaction
translate
translation
trim
true
unknown
upper
usage
value
varchar
varying
work

nchar	year
	zone

Appendix L: Returning to the Top Frame

This section contains the following topics:

For menu-driven applications that are more than two levels deep, it is important to provide a way for users to quickly jump from a deeply nested frame to the application's top frame. This appendix shows how you can use 4GL to implement this in an ABF or Vision application.

This method utilizes return codes to communicate between calling ("parent") and called ("child") frames. For the application user, selecting the menu item, Top, returns the application to the Top frame without displaying any intervening forms on the way up.

Example Frame Calling Sequence

In this example, Top calls Frame2, Frame2 calls Frame3, and Frame3 calls Frame4. This can be illustrated as:

```
Top
   Frame2
      Frame3
         Frame4
```

The application developer wants to allow the application user to return directly to the Top frame from Frame4 *without* displaying the forms for Frame3 or Frame2.

To return from Frame4, pass a known return code (say 11) to indicate "gotoTopFrame." Test for this return code in the calling routine and pass the same return code back to its calling routine, and so on, until the program has proceeded all the way back to the Top frame.

For example, to implement a new menu item Top in Frame4 that the application user can select, use this syntax:

```
Top =
begin
  return 11;
end
```

Frame3 receives Frame4's return code in the local variable Status via the following code fragment:

```
initialize (status = i4) = {}

Menuitem =
begin       /* Frame3 code that calls */
            /* Frame4 */
status := callframe Frame4 ;
if status = 11 then
  return 11 ;    /* Frame3 also returns */
else
  statements
endif;
end
```

When control passes back to Frame3 after Frame4 issues the return 11 statement, several things happen:

- The application passes a value of 11 into Frame3's hidden field, Status.
- In Frame3, processing resumes at the statement immediately following the callframe Frame4 statement. This is an if test on Status to check for a return code of 11.
- If the if test finds a value of 11 in Status, then Frame3 returns a value of 11 to Frame2, and Frame2 similarly returns 11 to the Top frame.

Embedded SQL Language Considerations

In the example above, use a callproc statement rather than a callframe statement. However, there are special considerations in embedded query language.

You must not return immediately from inside a display loop, a select loop, or from a routine that has cursors open. To exit each of these cleanly, use the appropriate embedded SQL statements before returning to the calling frame, as follows:

- Break out of a display loop with the breakdisplay or enddisplay statement.
- Exit from a select loop with the endselect statement.
- Close open cursors with the close statement.

For example, the following section shows a program fragment that illustrates returning a status of 11 from inside a display loop.

Example

```
exec frs activate menuitem 'Top';
exec frs begin;
    status = 11;
exec frs breakdisplay;
    /* drop out of display loop */
    /* Note: don't want to do   */
    /* 'return (11);' here because */
    /* we're inside a display loop */
exec frs end;
exec frs finalize;
    return ( status );
```

Appendix M: Runtime Error Processing

This section contains the following topics:

This appendix discusses two methods of handling errors.

By default, ABF and Vision applications print Ingres error messages on the standard output device. To customize error handling in your application, use one of the following methods:

- Use the 4GL statement inquire_sql to return error numbers to your application and code your application to take certain actions when particular errors occur. However, with this method, you cannot prevent an error message from printing to the user's screen.
- Write your own routine for handling selected errors. Declare your error handler to the FRS with the Ingres-provided routine called iiseterr(). Whenever an error occurs, Ingres invokes your error handler, passing the error number as an argument.

Standard inquire_sql Method

For ease of comparison between the iiseterr() method described in this chapter and the inquire_sql method, inquire_sql is described here. The syntax for inquire_sql is:

```
inquire_sql (error = errorno,
  rows = rowcount,
  text = errortext) ;
```

inquire_sql tells your 4GL program whether an error occurred during the last query or forms statement, the number of database rows affected by the statement, and the text of the error if it occurred in a query.

However, inquire_sql cannot prevent the display of error messages on the user's screen. By the time inquire_sql detects an error, the FRS has already displayed the error message.

How You Can Use iiseterr() in Ingres Applications

To implement a user-written Ingres error handler in an Ingres application, follow this step-by-step outline. See Errors That Cannot Be Handled in Error Handler Restrictions (see page 1344) for details about what errors are affected by the error handler.

The following example is written in embedded SQL/C, but you can write it in any other Ingres-supported language. Check the section, Runtime Error Processing, in the *Embedded QUEL Companion Guide,* for your language for details on how to use iiseterr() in other programming languages. iiseterr() is documented only in the EQUEL guides.

1. Write your error handler routine. This is an embedded SQL routine that receives error numbers as arguments, and contains your custom code for handling errors during the running of the application.

   Code your routine to return 0 for any errors you are handling locally. This suppresses the printing of error messages. For other errors, your routine must return the error number received. The error message is then displayed on the screen.

   For example, the following routine (named "myhandler") overrides the default error handling for error 4700 (deadlock). Error messages for this error are suppressed—only your 4GL program knows when they happen (via inquire_sql). Error messages for all other errors print on the screen at run time, as usual.

```
int myhandler (errno)
int *errno;

    {
    #define DEADLOCK 4700
int status = 0;

  switch (*errno)
      {
    case DEADLOCK:
      status = 0;       /* suppress message */
      break;
    default:
      status = *errno; /* All other errors:*/
                    /* print message */
      break;
    }
  return (status);
}
```

2. Write an embedded SQL routine that calls iiseterr() and passes it the name of your user-written error handler.

   For example, the following routine (named "seterr") declares the error handler (named "myhandler") to be used in an Ingres application.

   ```
   seterr ()
   {
     extern int myhandler();
     iiseterr (myhandler);
     return;
   }
   ```

3. Within your ABF or Vision application, create a procedure for each of the two embedded-SQL routines.

   In this example, create procedures named "myhandler" and "seterr."

4. In the initialize section of the application's Top frame, call the routine that declares the user-written error handler. This is the routine that you wrote in step 2.

   For example:

   ```
   /* Top frame of application */
   initialize ( ) =
   begin
     callproc seterr ;     /* Do before any */
                           /* database statements */
     statements
   end
   ```

   The application calls the procedure "seterr" when it starts up. The procedure "seterr" tells iiseterr() that the name of the user-written error handler for this application is "myhandler."

After your application calls iiseterr() and passes the name "myhandler," error handling is affected in all frames in the application except embedded SQL procedures. See Effect on Embedded SQL Procedures (see page 1344).

Interaction Between iiseterr() and inquire_sql

The user-written error handler and 4GL's inquire_sql statement never interact. The use of iiseterr() does not affect the error information or rowcount available to your program via the inquire_sql statement. Even if the error handler routine suppresses an error message by returning 0, the error number is still available to your 4GL program through the statement:

```
inquire_sql (errno = errno)
```

Error Handler Restrictions

Your error handler routine starts automatically after a numbered error occurs. In many cases, the query that triggered the error is still active. Do not issue any query statements from inside your error handler routine because the application occasionally "hangs" or returns runtime errors.

The only embedded SQL statements you can safely issue from your error handler routine are:

- message
- prompt
- sleep
- clear screen

Errors That Cannot Be Handled

The error handler called by iiseterr( ) handles numbered errors. It handles errors from the Ingres DBMS Server, the FRS, and the Ingres LIBQ; that is, errors known to the inquire_sql or inquire_frs statements. It does not handle 4GL statement errors or ABF/Vision runtime errors.

ABF or Vision starts the database before it sees your error handler declaration in the initialize section of your Top frame. Your application cannot "catch" database start-up errors. Neither your error handler routine nor the inquire_sql statement can catch these errors; Ingres simply issues an error message and your application exits.

Effect on Embedded SQL Procedures

In ABF or Vision applications, the user-defined error handler affects all frames and 4GL procedures in the application. However, any embedded SQL procedures that you include can use a different method of error handling, involving whenever statements and a communications area known as the SQLCA. See the *SQL Reference Guide* for more information about the SQLCA and the whenever statement.

When an ABF or Vision application calls an embedded SQL procedure that uses the SQLCA, your user-defined error handler is temporarily turned off. When the embedded SQL procedure returns control to ABF or Vision, your user-defined error handler resumes operation.

Multiple Calls

Your error handler routine is global to the application. It takes effect in every frame, 4GL procedure, and embedded query language procedure in your application.

Reset your error handler at any time, and as often as necessary during your application. For example, if a frame or procedure calls iiseterr() a second time, and passes it the name of another error handler, the new error handler replaces the one you defined previously.

Appendix N: Troubleshooting Guide

This section contains the following topics:

This appendix helps you solve some of the problems that you encounter when developing applications with ABF and Vision. These problems can occur in the following areas:

- Starting ABF or Vision
- Creating and editing applications
- Testing applications
- Creating an image of an application
- Running an application
- System performance

In each area, the appendix describes some of the most common problems and suggests courses of action to follow if you encounter these problems. In some cases, the appendix provides a way to solve the problem.

For other problems, consult your Ingres system administrator who, in turn, can call Ingres Technical Support. In these cases, the appendix provides instructions for defining the problem more narrowly, so that technical support can help resolve it quickly.

Before You Call Technical Support

If you are unable to solve a problem using the information provided in this appendix, the designated contact for your site can call Ingres Technical Support for help. Taking the following actions before calling can help your technical support representative solve your problem more quickly:

- Write down the exact error message number and text, and the time and date of the error
- Check the DBMS error log file (errlog.log) in the Ingres installation for any related DBMS errors

- Note in which frame or procedure the error occurred
- Use 4GL message statements to isolate the specific area of the code in which the error occurred
- If the error appears to be in a query:
  - Set the II_EMBED_SET logical/environment variable to printqry to see how the query is being issued to the DBMS.

    See the *System Administrator Guide* for more information on using II_EMBED_SET.
  - Try the query in the Terminal Monitor.
- Isolate the problem by:
  - Running the application in sections
  - Building a short application that contains code similar to the code that contains the error

Using Logicals/Environment Variables

Ingres uses the following types of Ingres environment variables/logicals:

Windows:

- Windows environment variables
- Installation-wide environment variables

Ingres environment variables are active only when you are using Ingres. Most are set in the symbol table and are visible only with the command ingprenv. A small subset can be reset by users in their local Windows environment.

UNIX:

- UNIX environment variables
- Installation-wide environment variables

Ingres environment variables are active only when you are using Ingres. Most are set in the symbol table and are visible only with the command ingprenv. A small subset can be reset by users in their local UNIX environment.

VMS:

- Group level Ingres logicals
- User-defined or process level logicals

Ingres logicals are defined in the appropriate logical name table.

When the procedures in this appendix require you to check the value of a logical or environment variable, they instruct you to use the appropriate commands to check each level where the command can be set. For detailed information about setting and using logicals and environment variables, see the *System Administrator Guide*.

Starting ABF or Vision

If you are unable to start ABF or Vision from the operating system command line or from the Ingres Menu, or if you experience problems immediately after start up, check the following areas.

The Ingres Environment

To run ABF and Vision, you must set up your Ingres environment correctly. See the section that applies to your operating system.

Windows: To check that you have defined the II_SYSTEM environment variable and that you have set up your directory path to include the location of the installation:

1. At the operating system prompt, type:

   ```
   set | find "II_SYSTEM"
   ```

   You must see a valid pathname.

2. At the operating system prompt, type:

   ```
   PATH
   ```

   You must see the %II_SYSTEM%\ingres\bin and %II_SYSTEM%\ingres\utility directory paths.

3. At the operating system prompt, type:

   ```
   dir % II_SYSTEM%\ingres\bin\vision*
   ```

   You must see the name of a file.

UNIX: To check that you have defined the II_SYSTEM environment variable and that you have set up your directory path to include the location of the installation:

1. At the operating system prompt, type:

   ```
   echo $II_SYSTEM
   ```

   You must see a valid pathname.

2. At the operating system prompt, type:

   ```
   echo $PATH
   ```

   You must see the $II_SYSTEM/ingres/bin and $II_SYSTEM/ingres/utility directories. If you do not see these directories, you must add them to your path.

3. At the operating system prompt, type one or both of the following commands.

   If you are running Vision, type:

   ```
   ls -l $II_SYSTEM/ingres/bin/vision
   ```

   If you are running ABF, type:

   ```
   ls -l $II_SYSTEM/ingres/bin/abf
   ```

   You must see the name of a file for which you have read and execute permission.

4. At the operating system prompt, type:

   ```
   ls -l $II_SYSTEM/ingres/bin/iiabf
   ```

   You must see the name of a file for which you have execute permission.

   If Steps 1, 3, and 4 do not produce the results described above, consult your Ingres system administrator.

VMS: To check that the IISYMBOLDEF.COM and IIJOBDEF.COM procedures have been executed:

1. At the operating system prompt, type:

   ```
   show logical ii_system
   ```

   You must see the name of a valid device or rooted directory. If you do not, contact the system administrator.

2. At the operating system prompt, type:

   ```
   show symbol vision
   ```

   or

   ```
   show symbol abf
   ```

   You must see the name of a valid command to execute the iiabf.exe program out of the II_SYSTEM:[INGRES.BIN] directory. If you do not, type the following command to define the necessary symbols:

   ```
   @II_SYSTEM:[INGRES.UTILITY]IISYMBOLDEF
   ```

3. Use the dir/security command to check that you have read and execute permissions for the file II_SYSTEM:[INGRES.BIN]IIABF.EXE.

Terminal Definition

The TERM_INGRES logical/environment variable must be defined correctly for your PC, terminal, or workstation, or you encounter screens incorrectly drawn, or menu and function keys operating incorrectly. To check that you have defined TERM_INGRES correctly, type the following command at the operating system prompt:

Windows:

```
ingprenv TERM_INGRES
```

UNIX:

```
echo $TERM_INGRES
```

VMS:

```
show logical term_ingres
```

You must see the appropriate terminal type for your PC, terminal, or workstation. For a list of valid terminal names, see the appropriate section of this guide.

Ingres Access Issues

Windows, UNIX, VMS: The following sections deal with access issues that can prevent you from starting ABF or Vision.

Servers

To use ABF and Vision, connect to a database, unless you use the –nodatabase flag. If you are connecting to a database, the necessary Ingres servers must be running. Use the show option of the iinamu utility to check that you have the appropriate Ingres servers running:

- The Name Server
- The DBMS Server
- The Communications Server, if you are using Net

If the appropriate servers are not running, ask the system administrator to start them.

See the *System Administrator Guide* for more information about the iinamu utility.

Permissions

ABF or Vision gives a start-up error if it cannot write the output files or temporary files that it requires. To check that you have read and write permissions to the appropriate directories, use the following procedures.

The ING_ABFDIR Directory

To check that you have defined the ING_ABFDIR logical/ environment variable to a directory for which you have read and write permissions:

Windows: At the operating system prompt, type:

```
ingprenv ING_ABFDIR
```

UNIX:

1. At the operating system prompt, type:

   ```
   echo $ING_ABFDIR
   ```

   If you see the name of a valid directory, proceed to step 3.

2. If you do not see the name of a valid directory, then type:

   ```
   ingprenv ING_ABFDIR
   ```

Use the ls -ld command to check that you have read and write permissions for this directory.

VMS:

1. At the operating system prompt, type:

   ```
   show logical ing_abfdir
   ```

   You must see the name of a valid directory.

2. Use the dir/security command to check that you have read/write/delete permissions for this directory.

If ING_ABFDIR is not defined correctly, contact the system administrator.

Temporary Files

ABF and Vision create temporary files in the directory defined by the II_TEMPORARY logical/environment variable, if you have defined it. Otherwise, temporary files are created in your current working directory. Use the following procedure to check that you have permissions to create temporary files:

Windows:

At the operating system prompt, type:

```
echo %II_TEMPORARY %

ingprenv II_TEMPORARY
```

UNIX:

At the operating system prompt, type each of the following commands as necessary, until you see the name of a valid directory:

```
echo $II_TEMPORARY

ingprenv II_TEMPORARY

pwd
```

Use the ls -ld command to check that you have read and write permissions for this directory.

VMS:

1. At the operating system prompt, type:

   ```
   show logical ii_temporary
   ```

   If you see the name of a valid directory, then proceed to step 3.

2. If you do not see the name of a valid directory, then type:

   ```
   show default
   ```

3. Use the dir/security command to check that you have read/write/delete permissions for this directory.

Creating and Editing Applications

If you are unable to create or edit a Vision or ABF application, check the following areas.

Creating Applications and Frames

If you are unable to create a new application or frame with the Create operation, check the application's source code directory and the ING_ABFDIR directory to make sure that the file system is not full.

Also, make sure:

- You are within the disk quota (see Source Code Directories in Editing Applications and Frames (see page 1354).)
- You have read/write/delete permissions for this directory (Use the procedures discussed in the section, Temporary Files, to check your permissions.)

Editing Applications and Frames

If you are unable to edit an existing application or frame with the Edit operation, read the following sections to try to locate the cause of the problem.

Source Code Directories

Determine the source code directory as follows:

1. In the Applications Catalog window, select MoreInfo from the menu.

   The MoreInfo about an Application window displays.
2. Select Defaults from the menu.

   The Application Defaults window displays. This window contains the name of the source code directory.

   Be sure that:

   - This directory exists.
   - It contains the source code files for the application.
   - You have read and write permissions for the files.

   If this directory is incorrect, change it in this window.
3. If you are using UNIX or VMS and your source directory has disk quota or permission problems, contact the system administrator.

Copying Applications

If you are editing an application that you copied in with the copyapp utility, check the source code directory as described above to make sure that it does not still point to the old location for the source code files. Also, ensure that you have copied the source code files.

System Editor

If you cannot use the system editor to edit source code files or Vision escape code files, take the following steps to make sure that you have defined the symbol ING_EDIT correctly:

Windows:

1. At the operating system prompt, type:

   ```
   ingprenv ING_EDIT
   ```

   The default editor command is displayed, if there is one set; otherwise, the default editor is edlin. Be sure the editor is in the PATH, or that the full path to the editor is specified.

2. If this is not the editor you want, change it with the following command:

   ```
   ingsetenv ING_EDIT editor_name [flags]
   ```

   In the above command, *editor_name* is the full pathname of the editor you want to use.

UNIX:

1. At the operating system prompt, type:

   ```
   echo $ING_EDIT
   ```

   The default editor command is displayed. Be sure the editor is in the PATH, or that the full path to the editor is specified.

2. If this is not the editor you want, change it with the following command:

   C shell:

   ```
   setenv ING_EDIT editor_name [flags]
   ```

   Bourne shell:

   ```
   ING_EDIT = editor_name [flags]
   export ING_EDIT
   ```

   In the above commands, *editor_name* is the full pathname of the editor you want to use.

VMS:

1. At the operating system prompt, type:

   ```
   show logical ing_edit
   ```

   The default editor command is displayed.

2. If this is not the editor you want, change it with the following command:

   ```
   define ing_edit editor_name / parameters
   ```

   In the above command, editor_name is the full directory specification of the editor you want to use.

Testing an Application

If you receive an error when you click Go to test a Vision or ABF application, check the following areas.

Compiling/Interpreting an Application

If you cannot compile an application when you click Go, the following sections can help you determine whether the 4GL compiler/interpreter is working correctly.

Disk Quotas

The 4GL compiler/interpreter creates files in various directories when you run an application. Use the following command or utility to make sure that you have not exceeded the disk quota or that the file system is not full:

Windows: Click Start, then Program, then Administrative Tools, to access the Disk Administrator.

UNIX:

```
df
```

VMS:

```
show quota/disk = directory
```

Use the previous command to check the following directories:

- The directory in which the compiler/interpreter creates the temporary files that it requires to run an application.

  This is the directory to which the II_TEMPORARY logical/UNIX environment variable is set, if any; otherwise, it is your current working directory.

- The directory in which the compiler/interpreter creates the C, object, and error listing files that it requires to run an application.

  This is the directory to which the ING_ABFDIR logical/environment variable is set.

- For Vision applications, the directory in which the compiler/ interpreter creates the 4GL source code files for the application.

Determine the source code directory as follows:

1. In the Vision Applications Catalog window, select MoreInfo from the menu.

   The MoreInfo about an Application window is displayed.

2. Select Defaults from the menu.

   The Application Defaults window is displayed. This window contains the name of the default source code directory.

Subprocess Quotas (VMS)

To check that you have a separate subprocess available as required by the compiler/interpreter, type the following command at the operating system prompt:

```
show process/quota
```

You must see a subprocess quota of at least 10.

If you encounter any quota or file system problems, contact the system administrator.

Compiling/Interpreting a Frame

If you are unable to compile an individual frame or procedure, check the following areas.

Source Code Files

Take the following steps to check the source code file for the frame or procedure:

- Try to edit the source code file for the frame to ensure that the file exists. Also, be sure that you have permission to access the file.
- Be sure that multiple frames or procedures do not share the same source code file.
- Ensure that the file system is not full for the source code directory. Also, use the procedure described above to ensure that you have not exceeded the disk quota.
- If the source code file or form is so large that the generated C++ code is too large for the C++ compiler, then break the frame up into smaller pieces.

Forms

Try to edit the form to ensure that the form exists. Also, be sure you have permission to use it.

Compilation Errors

If you run the frame or procedure and receive a message indicating compilation errors, use the Errors operation to determine the cause of the source code errors.

Frame Variables

If you are using a variable as a frame name in a callframe statement, the called frame cannot be compiled when you press Go. This occurs because ABF cannot tell at compilation time which frame to call at run time.

You must use one of the following methods to compile any frames that are called *only* through variable names:

- Use the Compile operation to compile the frame explicitly.
- Include code like the following in the source code for the frame that calls a frame by a variable name:

```
if 0 = 1 then
callframe framename1; /* one callframe for each frame */
callframe framename2; /*that could be called from this*/
callframe framename3; /* frame by callframe :variable */
endif
```

Because the above condition never can be met, the code never is executed; however, it is compiled. Remove the code after you have completed developing the application.

Uncompiled and Unnecessarily Compiled Frames

The 4GL compiler/interpreter compares the modification date of a frame's source code file with the date stored in the database to determine whether to recompile the frame. If a file's modification date is changed incorrectly, either of the problems in the following list can occur.

- The frame is compiled unnecessarily; this can happen when you:
- Copy the source code file to another location
- Access the source code file from multiple machines, or file servers whose clocks are not synchronized.
- **VMS:** Change the permissions on the source code file.
- The frame is not compiled; this can happen when you:
  - Restore into the directory an older version of the source code file
  - Call a frame that is specified as a variable in the 4GL source code
  - Access the source code file from multiple machines, or file servers whose clocks are not synchronized (ABF allows you a margin of error of approximately 10 minutes)

Linker Errors

If you are unable to link procedures that you have included in an application, check the following areas.

Link Options File

The link options file contains the names of object files for library procedures. Procedures that are called by the 4GL application, but that are not explicitly defined to the application, must be listed here. If you are running a 3GL procedure, take the following steps to ensure that the link options file exists and that it is defined correctly:

1. In the Applications Catalog window, click MoreInfo.

   The MoreInfo about an Application window displays.

2. Click Defaults.

   The Application Defaults window is displayed. This window must contain the name of the default link options file.

3. If no file name is displayed, check the ING_ABFOPT1 logical or environment variable, which also can point to the name of the default link options file.

4. Check that the link options file lists the appropriate object file for each procedure.

Interpreter Files

Ingres creates the following file in the ING_ABFDIR directory for applications that use 3GL procedures:

Windows: iiinterp.exe

UNIX: iiinterp

VMS: iiinterp.exe

This file contains a special version of the interpreter.

Because this file can be very large, check the directory to make sure that the file system is not full, or if you are using UNIX or VMS, that you have not exceeded the disk quota.

Symbol References

When referring to procedures in source code files, make sure that:

- You use the correct procedure name in any references in the source code file.
- You have defined to ABF or Vision all procedures called from 4GL.
- All library procedures exist in the library that you specify in the link options file.
- Multiple procedures do not have the same name.
- Sharable libraries do not have conflicting symbol names in the link options file (for UNIX or VMS).
- The libraries and object files specified in the link options file exist.

Creating an Image of an Application

If you are unable to create an image of a Vision or ABF application with the Image operation, check compiler problems in either of the following areas:

- Running the OSL compiler to process the 4GL code and forms into C++ files
- Running the C++ compiler to generate object code

Read the following sections to try to determine the cause of your problem.

4GL Compiler

If you cannot create an image of an application, check the following problem with your 4GL compiler. If you cannot create an image of an application, you check the following problems with your 4GL compiler. See the sections below that apply to your operating system.

Disk Quotas (UNIX and VMS only)

The 4GL compiler/interpreter creates files in various directories when you image an application. Use the following command to ensure that you have not exceeded the disk quota or that the file system is not full.

UNIX:

```
df
```

VMS:

show quota/device = *directory*

Disk Space (Microsoft Windows Only)

The 4GL compiler/interpreter creates files in various directories when you image an application. Use the following command to make sure that the file system is not full:

```
dir
```

Use the **dir** command to check the following directories:

- The directory in which the compiler/interpreter creates the temporary files that it requires to image an application

  This is the directory to which the II_TEMPORARY logical/UNIX environment variable is set, if any; otherwise, it is your current working directory

- The directory in which the compiler/interpreter creates the C, object, and error listing files that it requires to image an application

  This is the directory to which the ING_ABFDIR logical/environment variable is set.

- For Vision applications, the directory in which the compiler/interpreter creates the 4GL source code files

Determine the source code directory as follows:

1. On the Vision Applications Catalog window, select MoreInfo from the menu.

   The MoreInfo about an Application window is displayed.

2. Select Defaults from the menu.

   The Application Defaults window is displayed. This window contains the name of the default source code directory.

Subprocess Quotas (VMS only)

To check that you have a separate subprocess available as required by the compiler/interpreter, type the following command at the operating system prompt:

show process/quota

You must see a subprocess quota of at least 10dir.

Use the previous command to check the following directories:

- The directory in which the compiler/interpreter creates the temporary files that it requires to image an application

  This is the directory to which the II_TEMPORARY or CONFIG.ING environment variable is set, if any; otherwise, it is your current working directory.

- The directory in which the compiler/interpreter creates the C, object, and error listing files that it requires to image an application

  This is the directory to which the ING_ABFDIR or CONFIG.ING environment variable is set.

- For Vision applications, the directory in which the compiler/interpreter creates the 4GL source code files

Determine the source code directory as follows:

1. In the Vision Applications Catalog window, select MoreInfo from the menu.

   The MoreInfo about an Application window is displayed.

2. Select Defaults from the menu.

   The Application Defaults window is displayed. This window contains the name of the default source code directory.

C Compiler (WIN32)

If you cannot create an image of an application, check the following problems with your Microsoft Visual C++ compiler.

Working Files

The Microsoft Visual C++ compiler needs to create temporary files. Check to ensure that the file system is not full.

Consult your Microsoft Visual C++ compiler documentation to determine which directories to check and to ensure that the C++ compiler is installed properly.

Modifying Ingres Files

To create 4GL images or have interpreted 4GL call 3GL, modify several files contained in Ingres that assume:

- **II_SYSTEM=*x*:**
  where *x* >=**c**
- Microsoft Visual C++ compiler is in the PATH

These files are listed below in the following table:

File(s)	Comments
abfdyn.opt and **abflnk.opt**	These files in ingres\files must be identical. They assume: ▪ II_SYSTEM=*x*: where *x* is the current disk ▪ IIibce.lib (Microsoft Runtime Library) is on the current disk in \msc\lib. **Note:** If you set II_SYSTEM=*x*, you must change all the \ingres\lib and \msc\lib entries to reflect the disk drive *x* that you specified. For example, if II_SYSTEM=c:, you change \ingres\lib entries to c:\ingres\lib.
compat.h and **oslhdr.h**	These files in ingres\files assume that ingres\files is one of the directories in the environment variable INCLUDE=[directories].
utcom.def	This file in ingres\files assumes cl.exe (part of Microsoft C) is in the PATH. It also assumes ingres\bin is part of the PATH.
utlnk.def	This file in ingres\files assumes gluvii.exe is in the PATH. It also assumes that ingres\bin is part of the PATH. Add or remove options to gluvii.exe in the first line. For example: GLUVII -STACK 0x7000 -SEL x10000 -DPMI -VERBOSE@ **Note:** The period at the beginning of the first line must be present, and there must NOT be a period on the second or third line. The period indicates that the line is spawned directly rather than through command.com. Add or remove additional options for gluvii.exe in the file ingres\files\utlnk.glu.
***.bat**	These batch files in ingres\bin assume that ingres\bin is part of the PATH. To modify these batch files, remember that a DOS command line cannot be longer

File(s)	Comments
	than 126 characters.

Running Microsoft 6.00A and DOS 4.01

To overcome out of heap space errors that can occur if there is not enough memory to run Microsoft C 6.00A, use Rational Systems' OXYGEN with Microsoft C 6.00A.

C Compiler

If you cannot create an image of an application, check the following problems with your C++ compiler.

Working Files

If your C++ compiler or macro assembler needs to create temporary files, check the directory in which these files are created to make sure that the file system is not full and that you have not exceeded the disk quota.

Consult your C++ compiler documentation to determine which directories to check.

Running a Completed Application

If you are unable to run a completed ABF or Vision application, check the following areas.

Unknown Frames or Procedures

If you run an application from other than the default start frame (or if your application issues callframe or callproc statements), Vision or ABF cannot locate a particular frame or procedure. This is because Vision and ABF do not check that all required frames and procedures exist before running the application.

Also, if you are using a symbol in the link options file to call a procedure, ensure that you have defined the procedure correctly in your application.

Non-compiled Forms

Non-compiled forms are not part of an imaged application. Vision or ABF cannot locate a non-compiled form required by a running application if the form does not exist in the database.

If you are running the application against a database other than the development database, you must use the copyform utility to copy non-compiled forms into the production database.

Compiled Forms

If you are using a compiled form for a 3GL procedure, be sure to include the form file name in the link options file, or Vision or ABF cannot locate the form.

QBF Frames

QBF frames are not part of an imaged application; they always are retrieved from the database. If your application contains a QBF frame, be sure that:

- All JoinDefs used by the frame exist in the production database and are owned by you or the DBA for the production database
- All forms for QBFnames exist in the production database and are owned by you or the DBA for the production database

If necessary, use the copyform utility to copy the JoinDef or form from the development database to the production database.

Report Frames

Reports are not part of an imaged application; they always are retrieved from the database. If your Vision or ABF application uses report frames, be sure that all required reports exist in the production database and are owned by you or the DBA for the production database.

If necessary, use the copyrep/sreport utility to copy reports from the development database to the production database.

Queries

If your Vision or ABF application is unable to run the necessary queries correctly, check the following areas.

To check how queries are being sent to the DBMS, define the II_EMBED_SET logical/environment variable to "printqry."

All queries then are printed to the file iiprtqry.log in your current directory, or to the file specified with "set qryfile."

Queries Not Running

You cannot run queries against tables that you do not have permission to access. If you receive a "no grant or grant compatible permit" error message when you try to run a query, then your application is using such tables. To receive the proper permissions to use the tables, have the table owner or DBA issue an appropriate grant statement.

Queries Not Returning Data

If a query runs but returns no data, be sure that:

- You have specified all joins correctly
- You have not specified a qualification that eliminates all data
- Your select statements return data to the correct form or table field
- Your select statements do not return data to field or column names that differ from those on the form or table field, respectively
- You have not tried to select data to a hidden field or hidden table field column
- The database does not contain private user tables with the same names as tables that your application is using

Queries Returning Incorrect Data

If a query runs but returns the wrong data, be sure that:

- You have specified the correct qualifications
- The correlation names in your target list match those in the where clause
- You are not using repeat queries in a situation where such items as the target list or table names are being changed with variables
- The database does not contain private user tables with the same names as tables that your application is using

Cursors

If your application calls 3GL procedures that contain cursors, do not issue a commit statement during the database session. This causes all active cursors to close.

Global Variables

If your ABF or Vision application contains global variables, be sure that they are being initialized in or before the frame that calls them.

Initialize your global variables in the default start frame. However, if you run your application from another frame, your global variables are not initialized.

Parameters

If your ABF or Vision application passes parameters between frames or procedures, be sure that:

- You have used the byref keyword with any parameter whose value is to be changed when it is returned to the calling frame
- You are using the correct 3GL data types to receive any parameters that you pass to 3GL procedures
- Your parameter list for a call system statement does not cause the statement to exceed the limit for the operating system command line
- Your parameter list for a call report statement:
    - Does not cause the statement to exceed the limit for the operating system command line
    - Contains only quoted strings that conform to your operating system rules for quotes on command lines
    - Contains no unquoted dates
- Form field names match correctly for queries or forms that you pass as parameters

Table Fields

If you are having problems getting the correct data into or out of table fields, be sure that:

- Your table field manipulation statements (such as insertrow) correctly refer to the row number for the data window rather than the record number in the entire data set
- If you have defined a menu item that operates on the current row, then you have issued an inquire_forms statement to check that the cursor is in the table field before the menu item is activated
- You have not attempted to issue nested unloadtable loops; that is, a loop that starts another loop
- You have not selected so much data into a table field that your system slows down or even crashes because of inadequate memory

Keyboard and Function Keys

If users are having problems activating operations from the keyboard, you check the following areas.

Clearrest Key

If you do not want the Return key to clear the data from the rest of a field, set the TERM_INGRES logical/environment variable to a terminal mapping file that does not use the clearrest key. This is usually a setting that ends in "i;" for example, "vt100i."

Windows: If you do not want the Enter key to clear the data from the rest of a field, rename the following files in %II_SYSTEM%\ingres\files by typing:

```
rename ibmpc.map ibmpcold.map
```

```
rename ibmpcalt.map ibmpc.map
```

FRSkey Settings

Users can accidentally select menu items by striking a function key that does not have an associated menu item. This is possible because FRS key settings do not override default menu item key assignments.

For example, Function Key 7 activates the seventh menu item even if that menu item has an FRS key assignment that is displayed with the menu item text.

Expressions

If your application is not correctly evaluating expressions, be sure that you:

- Are not comparing string variables of different data types
- Have correctly considered trailing blanks in expressions
- Have not tried to make exact comparisons between floating point numbers used in computations; precision and roundoff can cause such comparisons to fail

Roles

If you are using roles in an ABF or Vision application and you are the DBA or a privileged user, run an application with an invalid role identifier.

In these cases, Ingres does not warn you that the role or password is invalid. Ingres issues an error message if a user without the proper privileges tries to run the application with an invalid role identifier or password.

Performance Problems

If your system performance is excessively slow when you are creating or running an ABF or Vision application, you check the following areas.

Prior Releases

If you are running an Ingres DBMS Server that is prior to this release, performance problems when editing, compiling, or imaging an application can occur. This problem is somewhat greater with Vision than with ABF, because Vision requires more information to be loaded from the data dictionary.

To alleviate this problem, type the following command at the prompt on the Ingres server system:

```
$ optimizedb -zk dbname -rii_objects -rii_abfdependencies
```

Note: The DBA can perform this command.

Slow-running Applications

Your application can run slowly because:

- You are selecting large amounts of data into table fields or arrays
- You are accessing very large arrays

Use the repeated keyword for queries that are run several times.

PART 6: Common Appendixes

Appendix O: Menu Operations

This section contains the following topics:

This appendix describes the menu operations available in these windows:

- Applications Catalog
- Create an Application
- MoreInfo about an Application
- Application Flow Diagram Editor
- Visual Query Editor

Each section of this appendix refers you to the appropriate chapter for information on using the operations listed.

The Applications Catalog Window

The Applications Catalog window provides operations to work with complete applications. For more information on using these operations, see Overview of Tools and Languages (see page 35) and Completing a Vision Application (see page 259) (except as noted in the following list). The menu operations are summarized as follows:

Create

Creates a new application. (It does not matter whether existing application names are highlighted.) Displays the Create an Application window.

Destroy

Removes the selected application. You must have authority to modify an application to remove it. You are prompted to confirm your decision.

Edit

Lets you change the selected application. Displays the ABF Edit an Application frame (ABF) or the application flow diagram (Vision).

Rename

Lets you change the name of an application. You are prompted for a new name. Press Return to cancel the Rename operation.

MoreInfo

Displays more information about the selected application on the MoreInfo About an Application frame.

Go

Runs the application. Use for testing the incomplete application or running the completed application. Go compiles the application (or parts of it) if necessary, then starts at the default start frame if one is defined. If not, you are prompted for a frame name.

Utilities

Display the Utilities submenu on the menu line in place of the Applications Catalog menu items. See Application Utilities Menu (see page 361), in the ABF part of this guide, or Using Vision Utilities (see page 243).

Help

Calls the online help facility

End

Returns you to the previous frame or the operating system prompt

Quit

Exits the tools and returns you to the operating system prompt

The Create an Application Window

The Create an Application window offers the following menu operations:

OK

Creates the application. Lets you define the frames by displaying the Edit an Application frame (ABF) or the Application Flow Diagram Editor (Vision).

Cancel

Cancels the Create operation and return to the Applications Catalog frame

ListChoices

Displays a list of choices for the current field from which you can select or a field description

Help

Accesses the Help Utility for this frame

The MoreInfo about an Application Window

The MoreInfo about an Application window provides the following menu operations to work with complete applications.

Next

Displays details on the next application in the Application Catalog list. ABF prompts you to save changes before moving.

Previous

Displays details on the previous application in the Application Catalog list. ABF prompts you to save changes before moving.

Save

Saves any changes made on this frame and to default settings on the Application Defaults frame

Defaults

Edits application-specific defaults. Defaults displays the Application Defaults pop-up frame, described in the next section.

Help

Accesses the Help Utility for this frame

End

Returns to the previous frame. Ends exits without saving changes. ABF prompts for verification if you select End without saving your changes.

The Application Flow Diagram Editor Menu

The Application Flow Diagram Editor provides the following menu operations to work with the frames in your Vision applications. For more information about using these operations, see Creating Frames (see page 87) (except as noted).

Create

Adds a new frame

Destroy

Destroys a frame

Edit

Lets you edit the form or change other features of the current frame (also see Defining Frames with Visual Queries (see page 117))

Compile

Generates and compiles the 4GL code and form for a frame and its tree

Go

Compiles the code and form and then runs a frame and its tree (also see Completing a Vision Application (see page 259))

Insert

Inserts existing frames into the application flow diagram

Remove

Removes frames from the application flow diagram

MoreInfo

Displays the MoreInfo About a Frame window that contains information about the frame

Utilities

Gives a submenu of utilities (see Using Vision Utilities (see page 243))

Catalog

Calls the Edit an Application (ABF Frame Catalog) window

Help

Calls the on-line help facility (see Getting Started in Vision (see page 77))

End

Saves your work and returns you to the Applications Catalog window

Quit

Saves your work and returns you to the operating system prompt

The Visual Query Editor Menu

The Visual Query Editor provides the following operations to define Vision Append, Browse, and Update frames. For more information about using these operations, see Defining Frames with Visual Queries (see page 117) (except as noted).

AddTable

Adds a Lookup table to the visual query

DelTable

Removes a Lookup table from the visual query

Edit

Lets you edit the form or various features of the frame definition

AddJoin

Adds a new join between tables

DelJoin

Removes a join between tables

NextTable

Moves the cursor to the first field of the next table in the visual query display

TableDef

Calls the Examine a Table window to display details about a table (see Using Vision Utilities (see page 243))

Utilities

Displays a submenu of Vision utilities (see Using Vision Utilities (see page 243))

ZoomOut

Displays a visual query in compressed mode

Help

Calls Vision's on-line help facility (see Getting Started in Vision (see page 77))

Cancel

Leaves an operation without saving any changes

End

Saves the visual query and redisplays the application flow diagram

Quit

Saves the visual query and returns you to the operating system prompt

Appendix P: Using Vision or ABF in a Multi-Developer Environment

This section contains the following topics:

This appendix discusses issues you must consider if you are sharing a Vision or ABF installation with other developers. These issues include:

- Starting Vision or ABF
- Sharing database objects—tables, forms, and reports—with other developers (Vision only)
- Using ABF in a multi-developer environment (ABF only)
- Managing locks in Vision or ABF

Starting Vision or ABF with Command Line Options

In some Ingres installations, you can access Vision or ABF in various ways to exercise specific privileges. To do this, include identifying parameters when you use the vision or ABF command to start Vision or ABF.

The full specification of the vision or ABF command is:

```
vision|ABF[nodename::]dbname [applicationname [framename]][-w][-5.0]
       [+wopen][-uusername][-Ggroupid]
```

nodename::dbname

Specifies the name of the Ingres database.

If you are using a database that resides on a remote host, you must specify *nodename,* followed by two colons. For example:

server1::orderdb

applicationname

Specifies the name of the application.

If specified, enter Vision at the Application Flow Diagram for that application or ABF at the Edit an Application frame for that application. If not specified, enter Vision or ABF at the Applications Catalog frame.

framename

Runs the application with the specified frame as the top frame

-w

Causes all warnings to be turned off. See Disabling Warnings (see page 1382) for more information.

-5.0

Invokes 4GL in a mode that is compatible with Ingres Release 5.0. See 5.0 Applications (see page 1383) for more information.

+wopen

Generates warnings if Vision or ABF detects statements that are not compatible with OpenSQL

-u*username*

Lets you run or edit an application as if you were the user represented by *username.*

Files created under this parameter are owned by the user actually running the Vision or ABF process, not by *username.*

To use this option, you must be a privileged user.

If you are using Ingres Enterprise Access products, see your Ingres Enterprise Access documentation before using this parameter.

-G*groupid*

Lets you run or edit the application as a member of the group specified

VMS: Capital letter parameters require double quotes in VMS, for example: "-G*groupid*"

Consult your system administrator to find out which parameters you can use.

Disabling Warnings

Whenever Vision or ABF compiles an application, it checks procedure names for conflicts with system function names. Vision or ABF also runs 4GL checks for some syntax errors. If conflicts or errors exist, Vision or ABF displays them. You must correct them to remove the warning. You can skip this check by placing the -w parameter on the vision or ABF command line.

5.0 Applications

Vision or ABF runs older applications with 5.0 semantic dependencies, but you must invoke Vision or ABF with the 5.0 parameter. Vision or ABF compiles the 4GL code with backward compatibility. Omit this parameter with newer applications.

Sharing Objects with Other Vision Developers

When you create an application or application component, such as a form, Vision recognizes you (that is, your account) as its owner. In this way, Vision protects the integrity of individual users' objects in a common database.

These guidelines describe how Vision handles the ownership of the following types of objects, and how you access another user's objects:

Applications

Any user can access an application, but cannot have access to the necessary objects to run the application.

Application objects, such as forms

Access any objects owned by the database administrator (DBA) for the database, if you do not have an object with the same name. Vision searches for an object owned by you first. If it does not find an object owned by you, then it searches the objects owned by the DBA.

You cannot access application objects owned by other users.

Tables and database procedures

You can access a table or a database procedure if the owner of the object (or another privileged user) has granted you the privileges to access them.

You must indicate the owner when you specify the table or procedure, unless the table or procedure is owned by the DBA and you do not have an object with the same name. For more information about object ownership, see the *SQL Reference Guide*.

If you are working with other developers, use one of the following methods to ensure that all Vision developers can share objects:

- If the DBA approves, log in directly to the DBA account before you start Vision.
- Alternatively, have the system administrator create a new user account. Each user then must log in to Vision through this account rather than a personal account.

  Any objects that a user creates in this account become available to all other users of the account. However, there is no way to tell what a specific user of the account does.

- If you have sufficient Ingres privileges, start Vision from your own account, assuming the identity of the DBA. To do this, use the vision command with the -u option:

  vision -u*dba dbname* [*applicationname*]

  dba

  Represents the DBA's username

  dbname

  Represents the database name. If you are using a PC, you must specify *nodename*::*dbname,* which is the name of the remote node on which the database is located.

  applicationname

  Is an optional application name

  The -u command option lets you create objects belonging to the user whose name you specify—in this case, the DBA.

- Create a master application owned by the DBA or another user. Developers can create objects in their own versions of the application, in their own accounts. The developers can then use iiexport and iiimport to merge the objects into the master application. The iiimport utility considers all imported objects to be owned by the owner of the master application.

Using ABF in a Multi-Developer Environment

When you create a database object, such as a table, form or report, Ingres recognizes you (that is, your account) as its owner. In this way, Ingres protects the integrity of an individual user's objects in a common database.

However, this also means that other developers cannot use your objects in their applications. For example, a frame that you create cannot use a table created by another owner.

Another developer can access an application that you created, but cannot use your tables or forms to run it.

You can access objects created by another user (that is, another account) only when those objects are created by the DBA for the database. All users can access objects owned by the DBA.

Therefore, use the following method to create objects that all ABF developers can share:

1. The system administrator creates a new user account, and then the system administrator creates a new user account with the same name.
2. Each developer accesses ABF by logging into this account rather than a personal account.

   Any objects that a user creates in this account become available to all other users of the account. Be aware that there is no way to tell what a specific user of the account does.
3. Each developer creates a separate application that corresponds to a tree of the main application.

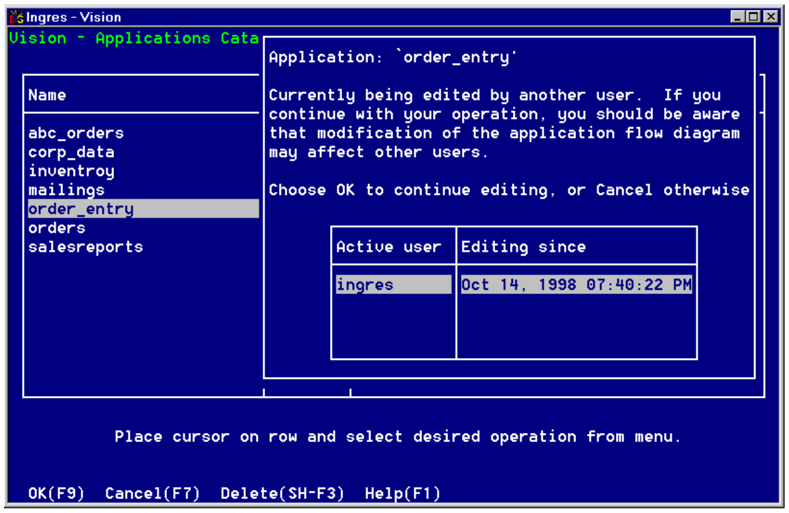
   Because each of these applications has a source code directory, you can avoid the problem of having ABF recompile frames that other developers have been editing.
4. Have one developer integrate the application components as follows:
   a. Copy all source code to a single directory.
   b. Create the main application.
   c. For each frame or procedure in each developer's application, create a frame or procedure in the main application with the same name, form, and source file.
   d. Recompile the application and create a new image.

Managing Locks in Vision or ABF

Vision or ABF warns you if another developer is using an application or frame that you want to access. Vision or ABF handles simultaneous access as follows:

- If you select Edit for an application from the Applications Catalog window, Vision or ABF checks whether another user currently is editing the application.

  Vision or ABF presents a pop-up, shown in the following figure, that lists the login names of any other current users in the application. You can continue to edit the application or cancel the edit session.



```
Ingres - Vision
Vision - Applications Cata

Name

abc_orders
corp_data
inventroy
mailings
order_entry
orders
salesreports

Application: `order_entry'

Currently being edited by another user.  If you
continue with your operation, you should be aware
that modification of the application flow diagram
may affect other users.

Choose OK to continue editing, or Cancel otherwise

Active user   Editing since
ingres        Oct 14, 1998 07:40:22 PM

Place cursor on row and select desired operation from menu.

OK(F9)  Cancel(F7)  Delete(SH-F3)  Help(F1)
```

  Before you continue to edit, check to see whether any changes you make to the application might affect other users (or whether their edits might affect you). When more than one user edits an application at the same time, Vision or ABF saves the last changes that are made.

  If you terminate a session unnaturally while editing an application—your system goes down or you abort a session—and you try to edit the application again, Vision or ABF treats you as a current user from your original session and from your new session.

  If this happens, you can ignore the warning, or select Delete to delete the lock. You also can use the LockStatus utility described in this appendix to delete the lock caused by your previous session.

- Vision refreshes the application flow diagram display every five minutes, looking to see if another user has changed the application flow diagram. If this has happened, Vision displays a pop-up prompt to let you refresh your version to incorporate the other user's changes.

  If Vision has found any changes, it displays this prompt immediately if you make a menu selection in the Application Flow Diagram Editor or after five minutes if you have not made a menu selection for that long.

  See Setting the Display Refresh Interval (Vision Only) (see page 1387) for information on changing the interval at which Vision refreshes the application flow diagram display.

- If you select Edit for a frame of an application that another user currently is editing, Vision or ABF displays a pop-up that lists the other user. You can continue to edit the frame or cancel the request.

  If you continue to edit the frame, be aware that concurrent users cannot see each other's work. Therefore, your changes can be overwritten by (or can overwrite) those of another user, because Vision or ABF saves only the last changes that are made.

- You cannot destroy an application or frame that another user is currently editing. If you try to do so, Vision or ABF displays an error message. This message contains the name of the other user of the frame.

Setting the Display Refresh Interval (Vision Only)

By default, Vision or ABF refreshes the application flow diagram display every 5 minutes (300 seconds). You can control this interval by setting the II_AFD_TIMEOUT logical or environment variable.

You can set II_AFD_TIMEOUT to any value from 0 to 300 seconds. A value of 0 causes the display not to be refreshed. A value of up to 20 seconds refreshes the display every 20 seconds.

Using the LockStatus Utility

Vision or ABF provides a utility to manage the locks that it creates in the situations described above. You can use the LockStatus utility to:

- View all the current locks in the database
- View detailed information about a specific lock
- Delete a lock (if you are the DBA)
- Refresh the display to reflect the most current locks

Each of these functions is described below.

Call the LockStatus Utility

To call the LockStatus utility

1. Select Utilities from the menu in any of these windows:

   - The Applications Catalog
   - Application Flow Diagram Editor (Vision only)
   - ABF Frame Catalog

   Because the LockStatus utility manages locks for the entire database, you can call it from any application.

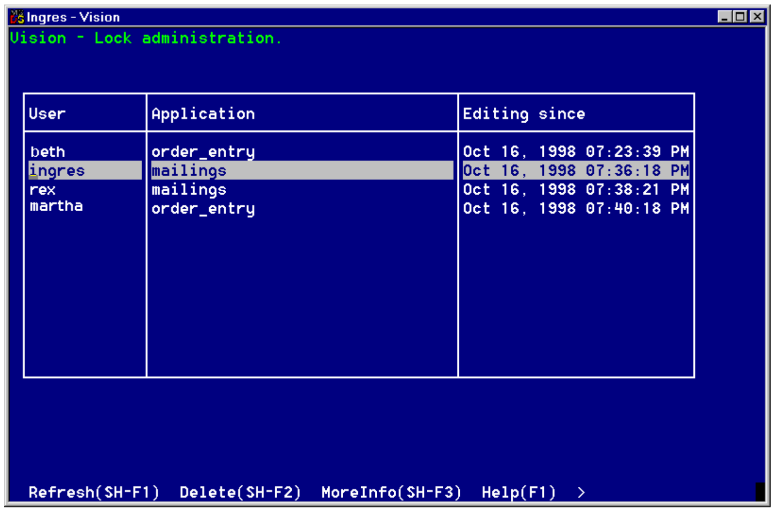
2. Select LockStatus from the submenu.

   Vision or ABF displays the Lock Administration window, as shown in the following figure. For each current lock, the display shows:

   - The name of the user who holds the lock
   - The application on which the lock is held
   - When the user began to edit the application

   This information is helpful to identify old locks that must be deleted.

After you display the Lock Administration window, you can use the available menu operations to perform functions described in the following sections.



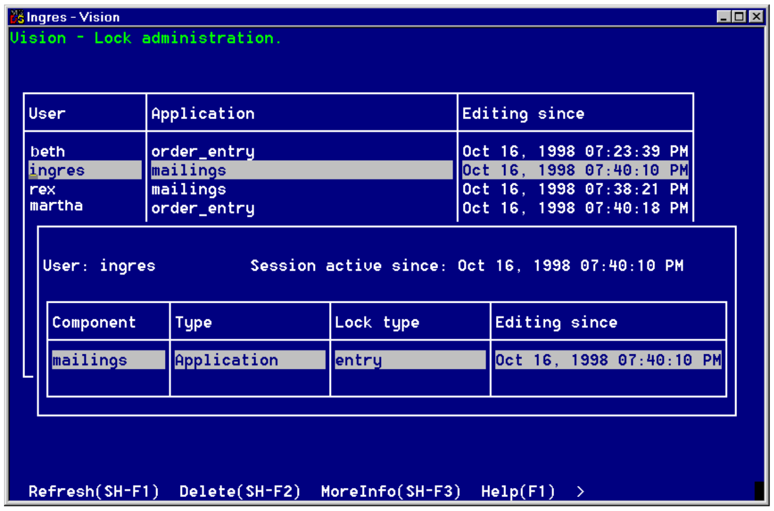
```
Ingres - Vision
Vision - Lock administration.

 User     Application              Editing since

 beth     order_entry              Oct 16, 1998 07:23:39 PM
 ingres   mailings                 Oct 16, 1998 07:36:18 PM
 rex      mailings                 Oct 16, 1998 07:38:21 PM
 martha   order_entry              Oct 16, 1998 07:40:18 PM

 Refresh(SH-F1)  Delete(SH-F2)  MoreInfo(SH-F3)  Help(F1)  >
```

Displaying More Information About a Lock

The Lock Administration window lists all application-level locks, but does not give details about each lock. For example, there can be two locks for the same application—one for the entire application, the other for a specific frame.

To display details about a specific lock, use the MoreInfo menu operation. The window displays the following information as shown in the following figure:

- The user who holds the lock
- When the user's session began
- *Component*—the name of the application or frame on which the lock is held
- *Type*—whether the component is an application or frame



```
Vision - Lock administration.

User      Application                Editing since
beth      order_entry                Oct 16, 1998 07:23:39 PM
ingres    mailings                   Oct 16, 1998 07:40:10 PM
rex       mailings                   Oct 16, 1998 07:38:21 PM
martha    order_entry                Oct 16, 1998 07:40:18 PM

User: ingres          Session active since: Oct 16, 1998 07:40:10 PM

Component   Type          Lock type     Editing since
mailings    Application   entry         Oct 16, 1998 07:40:10 PM

Refresh(SH-F1)  Delete(SH-F2)  MoreInfo(SH-F3)  Help(F1)  >
```

- Lock Type
  - An *entry lock* is simply a warning that another user is editing the component
  - A *write lock* prohibits you from editing the component
  - A *refresh* lock is similar to an entry lock; it also indicates that Vision or ABF is about to refresh the application flow diagram display
- When the lock holder began to edit the component

Deleting a Lock

You can use the Delete menu operation in the Lock Administration window to delete locks on an application or frame. You must be the database administrator to delete locks held by other users. You can always delete your own locks.

Before deleting a lock, be sure that your cursor is positioned on the correct session. Use the MoreInfo operation if necessary to view the lock in detail.

Refreshing the Current Lock Display

The lock display in the Lock Administration window is current at the time you display it. After five minutes, Vision or ABF redisplays the window to reflect any changes in the lock status.

You can use the Refresh menu operation to redisplay the window in less than five minutes. In this way, you can use the Lock Administration utility to monitor the activity of Vision or ABF users in the database.

Index